**Chapter 10. Designing a URL Shortener Service**

**A NOTE FOR EARLY RELEASE READERS**

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 13th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

We explored the concept of DNS in Chapter 9, noting that it’s easier to remember a website’s URL than their IP addresses—but what about the long URLs? It’s easy to remember the root part of the URL (for example, learning.oreilly.com in <https://learning.oreilly.com/library/view/learning-system-design/9781098146887/>) but you can easily forget the long URL. We often tend to embed [links](https://learning.oreilly.com/library/view/learning-system-design/9781098146887/) to text because that increases the readability. Another way of sharing long URLs is by shortening them to shorter URLs. For example, LinkedIn automatically shortens any URL that is part of a post because it helps in increasing post readability and user interaction. In microblogging applications such as Twitter (now known as X), there is a limit on characters for a single post , so to reduce the length of your text, you can shorten the URL via some URL shortener service such as [tinyurl](https://tinyurl.com/app" \t "_blank) or [bitly](https://bitly.com/" \t "_blank) and attach it to the post. This chapter explores the design of an URL shortener service and discusses how to deploy the system on AWS Cloud. We’ll start our discussion with requirement gathering and expectations from the system and later jump to the fine details of the system.

**System Requirements**

You should have a clear goal in mind of what problem you need to solve. From there, the next step to design any such system, big or small, is to gather the requirements. These requirements include:

* Why is this system needed? What is the business use case this system is solving?
* Who are the users of this system? How many users?
* Is there already a system that you can leverage instead of designing a new system from scratch? Systems like bitly expose URL shortening APIs to business customers so why should one build their own system instead of using bitly system APIs.
* Is the system’s latency critical?

The requirements list can be huge and time consuming but it’s a very important step. We had multiple discussions in Part I and Part II regarding how to compare two technologies and pick the best solution depending on your understanding of the business use case. It is not possible to weigh the pros and cons of your options if your requirements are not clearly defined. The requirements of any system are widely specified in two categories– functional requirements (FR) and non-functional requirements (NFR).

**Functional and Non-Functional Requirements**

Functional requirements mean functionalities or features offered by the system to the end users. The main expectations from a URL shortener service are simple:

* Systems takes input as a long URL and returns a shortened URL.
* The short URL should redirect to the long URL when accessed by any user.

Along with the most critical requirements described above, you can also consider value add requirements which can be nice to have from a customer’s perspective or helpful in deriving business value. Some of these include:

* Custom URL creation support.
* Analytics on the URL access patterns such as most popular URLs.
* Expiry for a URL so that the URL is auto expired and is no longer accessible after a fixed period of time.
* The application should be developed as plugin based architecture ensuring extensibility of the architecture. The system has capability to expose APIs to the third party clients to integrate their applications with our system to generate short URLs.

We should always gather as many requirements as possible about what might come in the future, but we shouldn’t get so caught up thinking about future expectations that we lose track of current expectations. The additional requirements will help to design a system in an extensible way so that new features can be added without needing to re-architect the system.

Another type of requirements are non-functional requirements, also called NFR. NFR determines constraints the system operates on and don’t directly impact the user feature wise, but are important as they ensure the quality of system operations. To take a few examples:

* Security, to ensure the system is not exploited by bad actors.
* High availability of the system, to ensure a high uptime percentage in a year.
* Observability, to ensure appropriate metrics/alerts are in place for constant monitoring of system health.
* Low latency for short URL creation and redirection.
* Datastores should be durable and ensure correctness of the data for the expiry time configured for an URL. The data should reside in the system until the expiry time or explicitly removed by a user.
* Fault tolerance in the system via mechanisms such as retry handling.
* Interoperability in the system architecture, as the system operates at high scale and can be broken down into multiple subsystems.How will different subsystems interact with each other?

These requirements will help in making better decisions while designing the system. Another aspect of the requirement gathering step is identifying the scale the system will operate on.

**System Scale**

Scale refers to the number of users or the traffic we’re expecting on the system. We should be able to answer questions such as:

* What is the potential number of user requests per second to generate a short URL?
* How many user requests for short URL to long URL redirection?
* Do we’ve any idea around the amount of storage we require to store the data?, etc.

Coming up with scale numbers can be tricky if you’re starting out fresh with any system and you might not have 100% accurate data to come up with these numbers. In these scenarios, it’s ok to make a fair assumption with a balanced thinking of not too little and not too much.

We recommend following the [Make it Work, Make it Right, Make it Fast](https://wiki.c2.com/?MakeItWorkMakeItRightMakeItFast) principle and all the chapters in Part III follow the same philosophy. It’s good to be optimistic that one day our system will serve one billion active users but you don’t have to build the system from day one to support this scale. The goal of huge scale should never hold you back from launching the product—just make sure that the system is extensible enough, so if there is a need in future, it can be evolved as new users are on-boarded and new features are introduced. For example, [Figure 10-1](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_1_amazon_com_initial_website_launch_from_1995) shows the initial launch of the amazon.com website—there was no point back then thinking of making it work for a billion customers.



**Figure 10-1. Amazon.com initial website launch from 1995**

**NOTE**

Some other reasons for redesigning any system could be the cost and operational maintenance such as when Uber started with DynamoDB to build a ledger store but later on moved to their own custom build storage solution, [DocStore](https://www.uber.com/en-IN/blog/dynamodb-to-docstore-migration/" \t "_blank).

The same principle applies while we figure out components in any system, the goal should always be to get the system out in the hands of the public and gather feedback. There is no win in over-optimizing the system if we never reach the goal of solving the customer problems and finally generating revenue from it; after all, everyone is in business to earn money from the designed system or help other people to do so (as with open source projects).

Coming back to the scale numbers our system should support, let’s move ahead with the following assumptions about our requirements:

Generate short URL from long URL - 1k requests/second

Short URL to long URL redirection - 20k requests/second

Average duration of URL persistence in system - 1 year

There are a lot of database solutions present in the market and in order to decide which one is best for our use case, there’s another important parameter we need to consider: total storage space required. Knowing this can help with answers to the questions such as

* What is the expected cost we’ll bear if we use Amazon DynamoDB or Amazon Aurora?
* Do we require data partitioning from the start?

Other things important for scale could be figuring out an instance type and number of instances for applications deployed on Amazon EC2 instances or memory requirements for AWS Lambda. As we pointed out earlier, it’s really hard to get the real numbers before putting the system in production and serving the actual consumer traffic. We recommend making fair assumptions and then improving further based on learnings. One key consideration is system load testing; Load testing helps in gaining confidence that the system is able to perform in the expected fashion (system is reliable) with the increased traffic demands. Let’s discuss the storage space requirements next.

**Storage Space**

The actual storage required will vary depending on the type of data storage solution we use, because the data storage and access patterns are different for each database. For an approximate calculation, the storage space will depend on the number of URLs, length of short URL & long URL, user metadata and URL metadata such as expiry time, creation time, etc. The number of URLs generated in a year can be derived from assumed traffic of 1000 requests/second.

Total URLs in 1 year = (1000 requests/second) \* 60 (per minute) \* 60 (per hour) \* 24 (per day) \* 365 (per year) = ~31.53 Billion

Before moving further along on storage calculation, we need to figure out the ideal length of a short URL. The length should be such that the system doesn’t run out of unique short URLs and should be able to support the required scale of 31.5 Billion unique URLs per year. In Chapter 9, we discussed the IPv6 addresses as a replacement for IPv4 addresses because the world is running out of unique IP addresses. We definitely don’t want this kind of problem to occur in our URL shortener system, so the length of the URL should be calculated with proper considerations from the beginning. You might suggest a length of 12 or 15 to be on the safe side, but that defeats the purpose of shortening the URL.

We can think about generating a short URL from the numbers 0-9, as well as a-z and A-Z alphanumeric characters. What is the maximum number of unique URLs generated for a given length with these 62 unique characters?

Length(1) = 62^1 = 62

Length(6) = 62^6 = 56.8 Billion

Length(7) = 62^7 = 3.5 Trillion

Length(8) = 62^8 = 218.3 Trillion

With reference to the calculations above, the URL system can support the required scale with URL length greater than or equal to 6. Considering the future scale, we can finalize URL length as 7, now let’s go back to our original question: what is the required storage space? Considering a simple database schema of storing short URLs, long URLs, expiry time and metadata, we can calculate our storage needs as:

Short URL(7 characters) = 7 bytes

Average long URL(100 characters) = 100 bytes

Expiration date(long) = 8 bytes

Average Metadata(userIP, userPreferences, etc.) = 1 KB

Total = ~1150 Bytes.

For simplicity, let’s take 1 KB as storage requirement for a single URL.

Total storage for 1 year = 31.53 Billion \* 1 KB = 29.37 TB

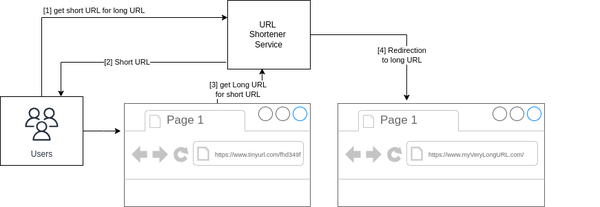
In addition to the above, the system has other storage requirements as well such as:

* Maintain cache to improve query performance.
* Analytics data.
* User authentication database. This can be required if the system offers additional capabilities to logged in users such as custom URL creation, view analytics, etc.

With all these things in our mind, let’s move forward with the design of URL shortener service.

**Starting with the Design**

We mentioned in the requirement gathering section about not thinking too far ahead because we don’t know what our system will look like in the future or what new features we might introduce to evolve with the market. We should try to keep our systems open ended so that we can introduce new features with ease when they are actually needed. [Figure 10-2](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_2_user_interaction_with_url_shortener_service) shows the most basic architecture and user interactions with the URL shortener service to meet the defined functional requirements. The users create a short URL corresponding to a long URL utilizing the URL shortener service and then use the same short URL in the web browser (or terminal) to access the URL. The URL shortener service figures out the long URL corresponding to the short URL and redirects the request.



**Figure 10-2. User interaction with URL Shortener Service**

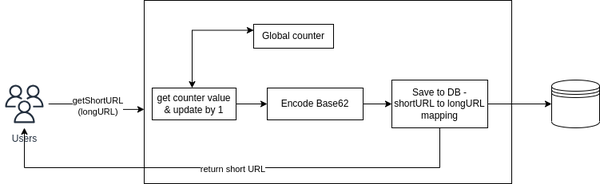
The first thing in the design we need to figure out is how a short URL can be generated from a long URL so as it can be optimally stored supporting faster user lookup queries.

**URL Shortening Algorithm**

Hashing seems to be a viable solution to solve this problem, using a hash function such as [MD5](https://en.wikipedia.org/wiki/MD5) which takes a long URL as input and returns a hash of this long URL as output. The length of the generated hash can be greater than 7, so the system should trim the generated hash to a reduced length. This solution is easy to implement but it has one problem– collision. Two different long URLs have a possibility of generating the same hash and even if the hash is different, the truncated length could be the same. At large scale, this can become a frequent problem, making short URL generation operation a bottleneck.

There can be different methods of collision handling—one possible solution is to take some other set of characters instead of truncating to the first 7 characters. The hash generation approach also requires a data store look up (multiple lookups in case of collisions) which adds to extra latency of the operation. In short, the hashing solution is not so perfect for our system requirements. Let’s think about an option that would avoid the collisions and perhaps also avoid the database lookups in order to make operation faster.

A key point to consider here is the long URL and short URL don’t have to be related—this would essentially be a unique Id generation system which can map the generated Id to a long URL. Let’s discuss a few approaches for generation of unique Id. The Id can be as simple as a counter incremented every time a new request is received from the users or an [AUTO\_INCREMENT](https://dev.mysql.com/doc/refman/8.0/en/example-auto-increment.html) like feature from a database. In case the system itself (without involvement of the database) handles the id generation, we’re sure of solving both the problems of collision handling and avoiding database lookups. Think of it as a server initializing a counter value with “1” and then increment it every time a new request arrives. The mapping can be stored to a database and then the short URL is returned to the users as shown in [Figure 10-3](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_3_short_url_generation_by_maintaining_a_counter). The sequence of alphanumeric characters from 0-9, a-z and A-Z when used for encoding is referred to as [Base62](https://en.wikipedia.org/wiki/Base62) encoding.



**Figure 10-3. Short URL generation by maintaining a counter**

Please note that by using Base62 encoding on the counter values (decimal numbers), the short URL will be unique but not always of length 7. For example:

Base62 (1) = 1

Base62 (10) = A

Base62 (61) = z

Base62 (62) = 10

Base62 (61) = 11

Base62 (1000001) = 4C93

To ensure the length is always 7, we can append random characters to the generated Base62 value from the decimal counter value. We haven’t decided on the maintenance of counter value. The simple solution is to maintain a global counter variable in our application which is updated after every new short URL generation. But there can be multiple machines on which the application is running which can lead to a duplicate counter value, and even if there is a single machine, it poses a risk of going down, which would make the counter value again start from zero.

To solve this, we can consider maintaining this global counter in a database. This way it solves the problem of a single machine going down (counter is not initialized to zero every time) or multiple machines being responsible for URL generation (duplicate counter values across the machines). However, it can also cause issues because we can’t allow multiple application threads to access the counter at the same time in order to avoid race conditions and if we apply a lock (a single machine reads and updates the value), it will increase operation latency. Let’s simplify the entire architecture further to ensure optimal application performance. We can assign additional responsibility to the system of pre-generating the short URLs since they are not dependent on the long URL. This way the multiple machines can utilize the pre-generated Ids (short URLs) solving all the bottlenecks.

Our URL shortener service consists of an additional component responsible for pre-generation of short URLs and keeping them in memory for faster access. An important decision to make here is whether we give this additional responsibility to our URL shortener service or host it as an independent service. We discussed monolith and microservices architecture in Chapter 7 and the benefits associated with them. A single component will ensure comparatively lesser hardware cost (at least in the beginning), as there’s no requirement of network calls between the services. What you need to consider in order to make the decision here is what value you’re getting out if this is an independent component instead of hosted as part of the main application. A separate id generation or key generation service (KGS) can be beneficial for other use cases as well and this is independent responsibility in itself doesn’t have to be tied to URL shortener service. Given this, we’ll go with the idea to host a separate service with the responsibility to generate unique keys and pass those onto our URL shortener service when requested. Let’s dive into that next.

**Key Generation Service**

KGS has a simple responsibility of providing unique Ids when requested by our URL shortener service. To ensure traffic control on KGS as well as make the URL shortener system more efficient, the operation can offer a list of Ids, say 1000 in single operation, and the URL shortener service can re-request once these Ids are exhausted. Now let’s dig a little deeper into the working of KGS—how should these unique Ids be generated?

We can consider the database auto increment feature such as MySQL [auto\_increment](https://dev.mysql.com/doc/refman/8.0/en/example-auto-increment.html" \t "_blank) or PostgreSQL [serial](https://www.postgresql.org/docs/9.1/datatype-numeric.html)/[sequence](https://www.postgresql.org/docs/8.1/sql-createsequence.html). This functionality is applicable for a leader instance (if multiple database instances) and it can become a single point of failure for maintenance of Ids, hampering the overall system availability. Further, implementing this feature becomes very hard to do in the case of multi-instance database setup such as horizontally scalable NoSQL databases. We can take inspiration from [Flickr](https://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/) architecture which uses MySQL auto-increment feature with REPLACE INTO query to get a globally unique new id on every new query as shown in code snippet below.

CREATE TABLE `Tickets64` (

`id` bigint(20) unsigned NOT NULL auto\_increment,

`stub` char(1) NOT NULL default '',

PRIMARY KEY (`id`),

UNIQUE KEY `stub` (`stub`)

) ENGINE=InnoDB

REPLACE INTO Tickets64 (stub) VALUES ('a');

SELECT LAST\_INSERT\_ID();

To increase system availability and avoid a single point of failure, two database servers are used starting with an even and odd number and offset as two to avoid collision as reflected in code snippet below.

TicketServer1:

auto-increment-increment = 2

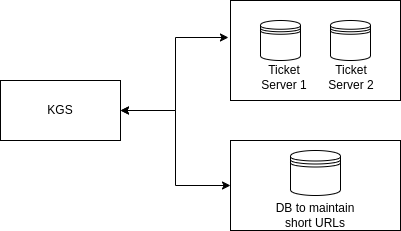
auto-increment-offset = 1

TicketServer2:

auto-increment-increment = 2

auto-increment-offset = 2

Further regular snapshots should be taken of these two databases as well to ensure backup is always kept in case of any fatal scenarios. As only a single row is maintained within a table, we can be sure that the server will not run out of storage space. The Id generation process speed will be dependent on database operation, so KGS should pre-generate the short URLs and maintain used and unused short URLs in another database, as shown in [Figure 10-4](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_4_unique_id_generation_via_ticket_servers).



**Figure 10-4. Unique Id generation via Ticket Servers**

The two ticket servers generate even and odd Ids respectively but this management overhead will increase if the ticket servers are deployed in multiple regions instead of just one. One potential solution to ensure uniqueness across regions could be adding a datacenter prefix to the short URL such as a 1 for us-east-1, 2 for eu-west-1 and so on, making the URLand make it 8 characters instead of 7 such as a 1 for us-east-1, 2 for eu-west-1 and so on.

**NOTE**

In 2010, Twitter proposed another approach for generation of 64-bit unique Ids, referred to as [Snowflake](https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake). [Snowflake](https://github.com/twitter-archive/snowflake/tree/snowflake-2010) can generate 4096 unique Ids per millisecond with a combination of timestamp (41 bits for epoch milliseconds timestamp), machine identifier number (10 bits giving us up to 1024 machines) and sequence number (12 bits for local counter per machine). The remaining one bit is a signed-bit always set to 0. The Snowflake approach will not directly work for our URL Shortener because the system always requires a 7 characters length string but implementation can be built using a similar concept. The main reason for Twitter to use this approach was the requirement for sorting and ensuring ordering guarantees. The URL shortener system has no such requirement of creating Ids based on timestamp so we’re free to choose the approach which is easy to implement and maintain the uniqueness.

Let’s quickly talk about database selection for KGS. KGS needs to maintain unique Ids which can be sent as part of response to the URL shortener service when requested. The KGS database should also ensure that it returns unused Ids on every new call and this information should be maintained in the database. Now the system has three options:

* Maintain two tables for unused and used keys. The server takes a list of keys from the unused table, updates the used table and returns a response. The maintenance of two tables for the same data offers additional challenges of data synchronization; It can be solved by using transactions across the tables via default database support or ensuring it via application code.
* Maintain a single table for both unused and used keys but maintain a distinction by keeping a boolean value.
* The table only keeps unused keys and deletes the keys from the database once they are sent to URL shortener service. This option has the least complexity, we don’t have any use case to lookup used keys but we recommend the system to still maintain the used keys for any use case that might come in future to support. You can consider this solution if the used keys are stored in some archival store and it can be better than the first approach in cost terms.

Additionally, KGS should keep some Ids in the buffer to serve the queries faster—but how many Ids should be in the buffer? A simple option is to generate all the short URLs in advance before we even launch the application, but we don’t recommend this. We suggest maintaining a balance—create unique Ids if they are needed, keeping some Ids in advance depending on application traffic. Let’s now discuss the APIs that need to be supported by the URL Shortener service and KGS.

**APIs**

APIs help clients integrate with the system to make use of supported functionalities. The URL shortener service should support two main APIs: creating a short URL and retrieving a long URL back from a short URL. The below code snippet shows an API signature for creation of short URLs.

POST /v1/createShortUrl

{

longUrl,

customUrl,

expiry,

userMetadata

}

The longUrl is a mandatory parameter in API requests for generating a short URL. The additional attributes are added to support other functionalities of the application. These include:

* customUrl parameter is helpful if the user wishes to generate a custom URL instead of a random 7 characters length string. For handling this parameter, the URL shortener service will directly check in the database if the custom URL is already created and if not, the mapping of custom URL to long URL will be stored in the database.
* expiry enables users to specify custom expiry on the URL instead of what is being defined by the system.
* userMetadata helps to gather extra information about the user such as IP address, geographical location, browser, etc helping in driving analytics.

**NOTE**

It’s possible that the same custom URL will be created by multiple users. Amazon DynamoDB’s default PutItem behavior is to update the item if the primary key already exists, but this is not a valid option for the URL shortener system. The PutItem request should contain [ConditionExpression](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.ConditionExpressions.html" \t "_blank) to fail the request for the same custom URL request.

The response of the API can be either success or failure. You might remember our discussion of HTTP status codes from Chapter 6—the status codes are helpful in determining the kind of HTTP response the client has received from the server. We should define proper mechanisms for both of them so that the clients can understand the response in failure scenarios and take any further actions such as retry. The failures can be due to invalid parameters from the client side or some issue at the backend application. Below are two example response scenarios:

HTTP/1.1 200 OK

{

"shortUrl": "https://www.tinyurl.com/dfjdf47"

}

HTTP/1.1 500 Internal Server Error

{

"error": "Please try again after some time"

}

As a user creates a short URL and shares with other people or accesses it themselves, the API request and response could look something like below:

GET v1/getLongUrl

{

shortUrl,

userMetadata

}

HTTP/1.1 302 Found

Location: https://www.example.com/my/long/url

HTTP/1.1 404 Not Found

{

"error": "short url doesn't exist"

}

Let’s discuss any additional considerations before moving to the AWS components being used to deploy the system in production.

**System Considerations**

By this point we’ve proposed two services in the architecture: the URL shortener service and KGS for supporting end to end functionality. The URL shortener service is responsible for short URL generation as well as redirection. The APIs are simple enough to be supported by a single system but the traffic patterns are quite different for both of these APIs; the traffic for URL redirection will be way higher as compared to URL generation.

Think of a celebrity generating a short URL and sharing the URL with followers on social media platforms—all the users will try to access the URL, increasing the overall system traffic. In these kinds of scenarios, we recommend starting with a single system and then evolving if traffic handling becomes a problem in a single system. Separating a system into two components can come with extra overhead, such as a database being accessed from two separate services. The URL shortener service will have both read and write use cases whereas the URL redirect service will have read use cases on the database.

It is not generally a good idea for two services to directly interact with a single database; the most robust solution is to expose the database operations via another service, responsible for handling all database operations along with cache maintenance.

We’ll also need to finalize database choice, given a wide variety of database solutions available in the market. As per the requirements, the system should essentially maintain a mapping between short URL and long URL to serve the URL redirect queries. This use case can be perfectly solved by using a key value database such as Amazon DynamoDB. There are multiple factors involved in the selection of a perfect database for a specific use case and we explored this idea in Chapters 2, 3 and 10. There are many benefits of DDB that we get out of the box, such as horizontal scaling, infrastructure management with no worry of scaling up read/write replicas, time-to-live configuration (TTL), and DDB streams for analytics..

Similarly, we can use DDB for KGS as well to maintain the generated Ids. DDB is a fully managed service by AWS and we as customers don’t have to worry about maintaining the storage infrastructure. You should list down the read and write query patterns to finalize on the DDB schema.

We’ve discussed the individual components and how the system will operate to serve the requirements. However, the business aspects of the application are important as well, and eventually we want to earn money from our systems. Some examples for revenue generation could be offering URL analytics or a custom URL domain for the paid users. Analytics in itself is a big portion of the URL shortener system and the entire end-to-end analytics pipeline is not in the scope of this book. However, in the next section we’ll shed some light on how custom domain support can be built in the system.

**Custom Domain Support**

One of the extended requirements of the system was to build an extensible system which is able to expose the APIs to third party clients. Assume that Google Drive leverages bitly services to shorten the URL but Google doesn’t want to share bitly short URLs to the users but rather a custom Google short URL.

Enabling custom domain support in the system is somewhat similar to maintaining multiple URL shortener systems, each with different domain names. Now the important point to consider here is whether the system has virtual separation or a physical separation. Physical separation means deploying multiple systems in parallel, each serving a specific domain, and virtual separation means the same system catering to all the clients with internal code separation or configurations.

Let’s try to solve this via a single DDB table which maintains the short URL to long URL mappings. DDB requires partition key and sort key as part of primary key, so the system defines:

Partition Key = short URL

Sort Key = tenant Id

The tenant Id is an identification useful in figuring out different tenants (systems onboarded to URL shortener system).

* The short URL is created via an API call by the onboarded system so they are expected to pass the tenant Id as part of the request. The tenant Id can be created when the system is onboarded or this can also be figured out based on credentials used by the system in the API call.
* On get call as well, tenant Id can be derived based on the domain being accessed.
* The key in the cache store should be created with a combination of short URL and tenant Id for easy retrievals.

We can definitely create different tables per tenant but a careful call should be taken based on the number of onboarded systems and their scale of operations. We can take a hybrid approach as well, often referred to as cell based architecture. Each cell is responsible to serve a set of onboarded tenants and the system maintains the mapping on which cell is responsible to serve a particular tenant and they can operate independently because they don’t share the data between them. There are also other things to be taken care of in this architecture like load balancing between the cells, one cell might just onboard a single tenant but another cell might include 10 tenants depending on the scale requirements.

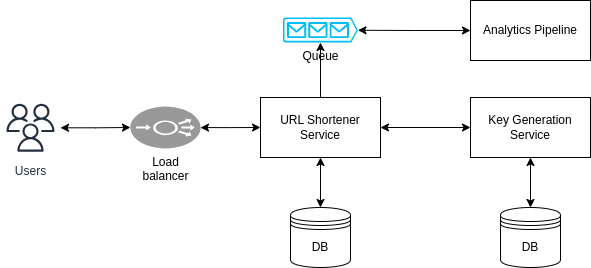
We discussed multiple components and considerations for the URL shortener system, now let’s combine the components to work together and discuss the launch of the product on AWS Cloud.

**Launching the System on AWS**

The entire system design is shown in [Figure 10-5](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_5_url_shortener_system_architecture). One additional component in the design which we haven’t yet discussed is the analytics pipeline. The URL shortener service publishes the event of user activity to the queue and the analytics pipeline takes care of analyzing and offering insights on data. For creation of URLs, users will not create short URLs directly—rather there will be a frontend application gathering the customer request and then forwarding to the URL shortener service for URL generation.

**NOTE**

To simplify the diagrams, the services introduced in our system design diagrams may not include a load balancer or API gateway to front the application. You can always assume that the service includes a load balancer or any similar component to distribute load into multiple instances of the application.

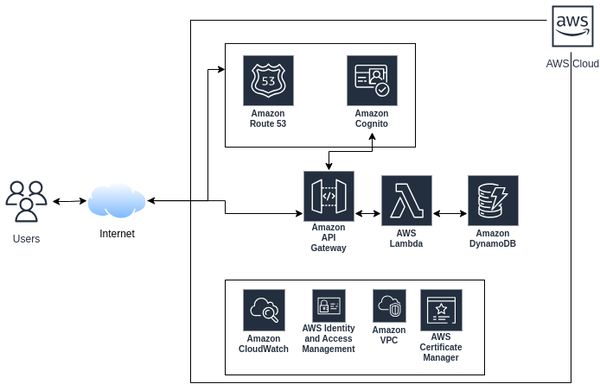


**Figure 10-5. URL shortener system architecture**

Now let’s deploy this architecture on AWS cloud starting from the initial launch.

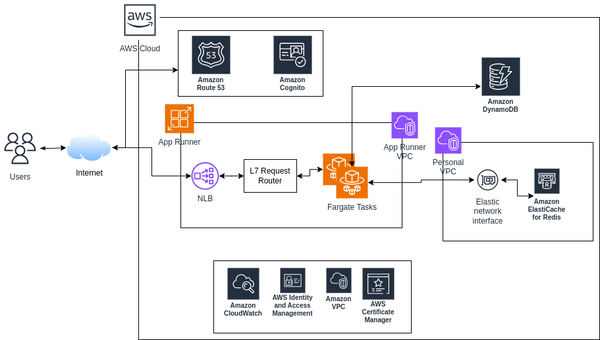
**Day Zero Architecture**

Keeping the [Make it Work, Make it Right, Make it Fast](https://wiki.c2.com/?MakeItWorkMakeItRightMakeItFast) principle in mind, the day zero architecture could look something like [Figure 10-6](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_6_url_shortener_service_on_day_zero).



**Figure 10-6. URL shortener service on day zero**

As we’re proposing to use AWS Lambda for operations on the URL, we can choose to deploy separate lambdas based on APIs or a single lambda handling all the operations. The diagram includes some additional services such as Amazon CloudWatch for monitoring, Amazon Cognito and AWS IAM for authentication and authorization, Amazon VPC to launch the resources and AWS Certificate Manager to SSL/TLS certificate management. Another approach for application deployment can be usage of [AWS App Runner](https://docs.aws.amazon.com/apprunner/latest/dg/architecture.html) service to fully manage deployment of containerised web applications or backend API services as shown in [Figure 10-7](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_7_day_zero_architecture_with_aws_app_runner). AWS App Runner launches all the resources in the VPC managed by AWS itself.



**Figure 10-7. Day zero architecture with AWS App Runner**

AWS App Runner abstracts all the components and customers just need to focus on application development. It is built with a combination of ECS Fargate, auto scaling, ELB and ECR and supports Python, Node.js, Java, Go, Rails, PHP and .Net as language runtime. Customers are required to specify a GitHub source code repository or an ECR image and App Runner takes all the responsibility of application deployment as ECS Fargate tasks. App Runner [CreateService](https://docs.aws.amazon.com/apprunner/latest/api/API_CreateService.html" \t "_blank) API returns a secure URL which can be used to access the service APIs.

Looking at the overall architecture, the user request is routed to NLB after domain resolution via Route 53 and then forwarded to Application Layer L7 request router which further redirects to ECS Fargate tasks. We additionally added Amazon ElastiCache in the diagram to showcase the [connectivity](https://aws.amazon.com/blogs/aws/new-for-app-runner-vpc-support/) between AWS App Runner VPC and customer managed VPC. AWS App Runner can also be configured to accept traffic via customer managed personal VPC by using AWS Private Link endpoint.

There is always a possibility that the day zero architecture system faces some bottlenecks as it scales to serve more user traffic and features, so let’s try to dig deeper into the architecture.

**Scaling to Millions and Beyond**

With our day zero architecture, we try to launch the product without planning too far ahead and choose the technologies we’re most familiar with instead of trying out something very new which will include a learning curve. In short, time to market should be minimum. Let’s consider an example of a compute platform. There are multiple options such as AWS Lambda, Amazon EC2, Amazon ECS, Amazon EKS. If you’re most comfortable with Amazon EKS and have prior experience with it, choose that option as a compute platform to start with.

We recommend setting proper metrics and alarms to identify any bottlenecks via tools such as Amazon CloudWatch and [AWS X-Ray](https://docs.aws.amazon.com/xray/latest/devguide/aws-xray.html). AWS X-Ray service helps in request tracing in distributed system architecture and figuring out which specific component is taking more time in processing. As we identify specific components causing issues in the overall system, we can dig deeper to figure out a resolution. We’ll dig deeper into the architectures in [Figure 10-6](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_6_url_shortener_service_on_day_zero) and [Figure 10-7](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_7_day_zero_architecture_with_aws_app_runner) to identify any issues and address them to ensure the system’s high availability.

**Storage Layer**

The system is using Amazon DynamoDB as a data store to maintain the URL mappings. We don’t have to worry about the scaling capabilities of DDB as it is managed completely by AWS. To begin with, we can utilize on-demand capacity mode and as we figure out traffic patterns, provisioned mode with auto scaling can be used to reduce costs. The application can be launched in multiple AWS regions and DDB global tables can be used to ensure data is replicated across the regions. A key consideration on global tables is approximately [one second](https://aws.amazon.com/blogs/database/part-2-build-resilient-applications-with-amazon-dynamodb-global-tables/) of replication lag between two or more regions, so it might happen that a short URL is not available in another region as soon as it is created.

For URLs generated by popular users, the read traffic can be really huge so there is no point in serving all read queries directly from the database. DDB guarantees single digit millisecond latency on direct key operations but to make it even more seamless and bring down latency to microseconds, we can add a caching layer on top of DDB such as Amazon ElastiCache or Amazon DynamoDB Accelerator (DAX) to improve read performance.

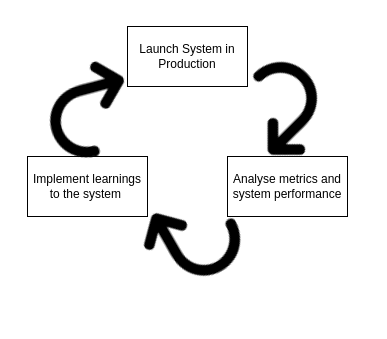
An important key consideration while you’re designing any system with DDB as your storage option is justifying the requirement for caching with DAX or ElastiCache—which one should you use? The ideal choice may be DAX as it is specifically built for DDB to improve query read performance, but Amazon ElastiCache might be a better choice if the queries are already being served by it for any other use case or you’re more familiar with the technology. This saves integration effort into newer technology, saving implementation bandwidth. Further it is always a good idea to perform benchmarking tests to compare two technologies to support a technical decision because in the end, we need a combination of low latency and a cost efficient solution.

We’ll focus next on the compute layer from the day zero architecture.

**Compute Layer**

We preferred AWS Lambda in [Figure 10-6](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_6_url_shortener_service_on_day_zero) and AWS App Runner in [Figure 10-7](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_7_day_zero_architecture_with_aws_app_runner) to avoid the setup of Amazon EC2 and avoid operational burden, but can Lambda or App Runner scale to millions of customers? AWS Lambda automatically scales as per traffic requirements without any customer intervention. However, the problem which we might face is p100 latency which can increase from time to time due to cold start issues. We discussed a couple of solutions to overcome the cold start problem in Chapter 11, such as provisioned concurrency which can help to address the latency bottleneck.

AWS App Runner allows maximum 25 instances per service and maximum 200 concurrent requests per instance, leading to 5000 concurrent requests per service. We recommend following the [documentation](https://docs.aws.amazon.com/general/latest/gr/apprunner.html) on AWS limits for updated information on service quotas. The number of requests limit can become a bottleneck as per our scale considerations in the System Scale section. A point of consideration here is the limits can be increased in the future by AWS and this should limit us in launching the first version of the product. AWS Cloud offerings or open source solutions evolve and cloud providers innovate on the behalf of their customers, so in the future it is possible that AWS App Runner will be able to support much higher scale with lower cost. Remember that no architecture is considered the final architecture and we can always reevaluate and reiterate in case of any bottlenecks as per the software lifecycle in [Figure 10-8](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_8_software_lifecycle).



**Figure 10-8. Software lifecycle**

Here are few things to consider to scale the compute part of your system:

* Consider migrating to unmanaged versions of services, such as moving to Amazon ECS EC2 from Amazon ECS Fargate to have more control over hardware configurations.
* The system will have auto scaling in place to handle any increased load during peak times. The ASGs should be in different AZs to withstand AZ downtimes. In the case of managed systems like Amazon ECS Fargate, AWS takes care of deploying the system into multiple AZs to ensure availability. We recommend taking this point into consideration if you’re managing the systems.

Let’s incorporate all these concepts and present the final architecture on AWS.

**Day N Architecture**

We’re using a single AWS account to host all the services in the architecture—this can be evolved further as necessary in the future to host services in different AWS accounts or to take any other approach of account separation as we discussed in the ‘Getting Started with AWS’ section of Chapter 9. Please note that using multiple AWS accounts can increase data flow cost across the services and shared databases can become difficult to manage, though it can make the responsibility of each account simpler with clear boundaries.

Expanding the discussion from Chapter 9, another degree of isolation is at the VPC level. If all the applications are created in the same AWS account, are they present in the same VPC or different VPC? Further, if in the same VPC, are they hosted in the same subnet or different subnets? The resources should definitely be launched in different subnets (even for a single application) to ensure high availability across the AZs. Launching all applications in a single VPC helps in reducing network costs, but needs extra care with managing the security of inter application communications. The rules can be managed by NACLs, and we recommend maintaining a limited number of rules. The rules are evaluated one by one and a large number of rules can compromise network efficiency by taking more time in evaluation (as compared to all rules being evaluated in parallel).

Our URL shortener system architecture can include five microservices for serving simplistic functional requirements:

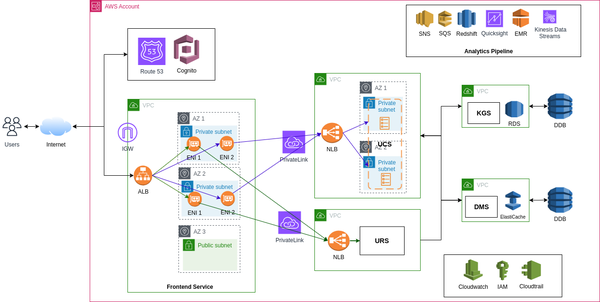
* FrontendService (FES) responsible for receiving the customer traffic and redirecting to specific service VPC as per the operation.
* URLCreatorService (UCS) responsible for creation of URLs.
* URLReaderService (URS) responsible for URL redirection from short URL to long URL.
* KeyGenerationService (KGS) responsible for generation of unique keys to be used as short URLs.
* DataManagementService (DMS) is a thin layer of CRUD APIs between databases and services in the architecture. DMS is introduced in the architecture because the DDB and ElastiCache are accessible to two systems.

We are looking at a huge scale so we recommend deploying the applications in separate VPCs for proper isolation. Another architecture pattern followed by organizations is deploying multiple microservices on the same EKS cluster (in the same VPC). This makes sense at limited scale but again in context of day N architecture, we recommend using [separate EKS clusters](https://aws.amazon.com/blogs/containers/multi-tenant-design-considerations-for-amazon-eks-clusters/) for service deployments for complete resource isolation (please revisit chapter 7 for kubernetes architecture).

**NOTE**

It is not always true that the AWS cloud components can introduce bottlenecks with increasing scale—it could also be the application code introducing bugs and bottlenecks in the system. For example, the DDB key schema worked well on day zero but is not performing very well with increased traffic. We can use tools such as [Amazon CodeGuru Reviewer](https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html) and [Amazon DevOps Guru](https://docs.aws.amazon.com/devops-guru/latest/userguide/welcome.html) to identify potential bottlenecks and then work towards resolution.

[Figure 10-9](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch10.html#fig_9_url_shortener_system) shows the final picture of the architecture of the URL shortener system.



**Figure 10-9. URL Shortener System**

Here are some additional points to keep in mind, which we left out of the diagram to make it cleaner:

* The services residing in different VPCs require a mechanism to connect them to each other. AWS Transit gateways (TGW) provide bidirectional communication between the VPCs while AWS PrivateLinks provide unidirectional communication. Further, the CIDR blocks of VPC should not overlap in case of TGW but PrivateLink doesn’t take this into consideration.
* The microservices also interact with AWS services which are present in our owned VPC. These services should be accessed via [AWS PrivateLink](https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/networking-connecting-vpc.html) instead of NAT gateways. This ensures the traffic doesn’t go over the internet which adds to application security and minimizes operation latency. NAT gateway should only be used in case an explicit connection to the internet is needed.
* Other ways for microservices running on Amazon ECS to establish communication with each other can be via [Amazon ECS Service Discovery](https://aws.amazon.com/blogs/aws/amazon-ecs-service-discovery/), [AWS App Mesh](https://aws.amazon.com/blogs/containers/connecting-services-across-multiple-accounts-using-aws-app-mesh-and-amazon-ecs/) and [Amazon ECS Service Connect](https://aws.amazon.com/blogs/aws/new-amazon-ecs-service-connect-enabling-easy-communication-between-microservices/).
* Amazon ElastiCache is used as a caching layer on top of DDB to serve read queries faster. ElastiCache will remove the least used data as per the cache eviction policy—we can introduce an additional AWS Lambda function (invoked on DDB data deletion event) to ensure data removal from the cache if removed from DDB. This can happen in the case of any custom expiry on the short URL by the users. Instead of the AWS Lambda function, we can also implement a consumer in DMS service itself which can handle this responsibility.
* The architecture presents deployment in a single AWS region. We recommend creating all the infrastructure via code for easy replication to other regions if required. The system resources are deployed in multiple AZs to ensure increased redundancy in case of AZ downtimes.
* The main reason for separating UCS and URS is the huge difference in traffic patterns. A separate service ensures independent scaling and clear separation of concern.
* The system architecture should ensure the increased traffic is from actual users and throttle any unwanted traffic at API Gateway or ELB level by utilizing AWS services such as [AWS WAF](https://docs.aws.amazon.com/waf/latest/developerguide/waf-chapter.html) and [AWS Shield](https://docs.aws.amazon.com/waf/latest/developerguide/shield-chapter.html) or deploying custom solutions. This can be handled in the Frontend service.

The points above are not an exhaustive list of issues to take care of while serving customers. We will cover additional topics in more detail in the remaining use case chapters of Part III and you should take a holistic view of all chapter learnings to deploy systems in production to ensure high standards.

**Conclusion**

We started off Part III with the URL shortener system which is a popular interview problem and a very common system we use in our daily lives. The gist of any new system design is to start with requirement clarifications and system boundaries. You should clearly know what a system is expected to do and what a system will never do—this helps in making better design decisions. Once the requirements and scope of the system is finalized, we move on to identifying the high level components. The scale a system is supposed to serve is an important factor to identify these components along with the business use case. A system operational with 10k users might not work the same way with 100k users.

We moved through multiple possible approaches, discussing their pros and cons, along with potential AWS Cloud components. We brought in many of the different AWS services we discussed in Part II of this book and we acknowledge that recognizing which one to choose for your specific use case could be difficult. We recommend choosing a technology that will take minimum time to understand and deploy—down the line, it can be modified or replaced by another technology if needed.

System design is an iterative process, so start small and enhance your systems on the go as needed. We recommend not thinking too far ahead, because your priorities may shift after five or ten years. In the next chapter, we’ll design a web crawler and search engine system such as Google search, using AWS components.