* **The size of the URL list could grow infinitely. How might you scale this beyond the memory capacity of the system?**

  If the size of the URL list grows infinitely, I might scale this size of the URL list using sharding. Sharding is a great option for partitioning database that data can be distributed across a number of servers.

  In Redis, data sharing splits all data across multiple Redis instances so that every instance will only contain a subset of keys. Partitioning in Redis allows for much larger databases, using the same of the memory of many computers, and allows for adding and removing Redis instances without any server downtime.

* **Assume that the number of requests will exceed the capacity of a single system, describe how might you solve this, and how might this change if you have to distribute this workload to an additional region, such as Europe.**

  I might solve this issue with a load balancer that distributes network traffic across a number of servers. Load balancer is used to increase capacity and reliability of systems. It improve the overall performance of applications by decreasing the workloads on servers and prevents any system server from becoming a single point of failure.

  If a single system demand increases, new servers can be easily added to the resource pool, and the load balancer will immediately begin sending traffic to the new server. Global server load balancing is one of the core capabilities that extends the core layer 4 and layer 7 capabilities so that they are applicable across geographically distributed server farms such as Europe.

* **What are some strategies you might use to update the service with new URLs? Updates may be as much as 5 thousand URLs a day with updates arriving every 10 minutes.**

  In order to update the service with new 5 thousand URLs a day with updates arriving every 10 minutes, I might build another service that only controls updating new URLs. If HTTP requests GET and PUT methods are separated into single API service to transfer data from client to server in HTTP protocol, then this might reduce the possibility of downtime, and more HTTP PUT requests will be sent through bandwidth.

Another strategy is prioritization of network traffic. The requests are all sent to the server as soon as they are discovered along with some prioritization information to let the server know the preferred ordering of the responses. Then, the server will deliver the most important responses first, followed by lower priority responses. This strategy is ideal for bandwidth utilization to reduce congestion.

* **You're woken up at 3am, what are some of the things you'll look for?**

  Server uptime is the first thing I will check at 3am. Then I might check other metrics for measuring the performance of servers: CPU, disk & memory utilization, throughput rate, latency, error logs. These server performance factors show the status of machine in the network and the amount of load processed by the network server.

* **Does that change anything you've done in the app?**

  At the beginning, I was planning to implement both of HTTP GET and POST methods in the application. As mentioned in the previous question, if 5 thousand URLs updates a day with updates arriving every 10minutes, then performing HTTP GET and POST methods at the same time in the application is not a good way to maintain the server. The server might experience increasing the hit and memory load within the limited resources, and may result in server downtime. Thus, I decided that this application controls only HTTP GET requests.

* **What are some considerations for the lifecycle of the app?**

  It is important to manage the entire lifecycle from the idea conception through the retirement of application.

  A solid application governance, development and operations are desirable to maintain coherence in software development. These three elements encompasses the initial scoping and requirements, application design, deployment, versioning, operational maintenance and ultimately, the end of life of the application.

* **You need to deploy new version of this application. What would you do?**

I would choose a rolling strategy to deploy new version of this application. A rolling deployment slowly replaces instances of the previous version of an application with instances of the new version of the application. This strategy typically waits for new pods to become ready via a readiness check before scaling down the old components while servers have no downtime during an application update, and the application supports having old code and new code running at the same time.