Scaling Stateless Components

Teacher Version

Software Architecture

March 29, 2023

Evan Hughes & Brae Webb

1 This Week

Our goal is to scale out the stateless component of our TaskOverflow application across multiple compute instances. Specifically we will need to:

- Route traffic to our deployed TaskOverflow application with a load balancer.
- Scale out TaskOverflow instances with autoscaling.
- Check the status of our instances with a healthcheck.
- Dynamically scale our application based on load.

2 Load Balancers

Load balancing distributes a load over a set of resources. For example, balacing network traffic across several servers. Load balancing is crucial to the scalability of modern systems, as often, one physical device can not process the large amount network traffic for (e.g.) a large website.

A service which load balances, is called a **Load Balancer**.

2.1 Routing Algorithms

A load balancer can implement many techinques to select which resource to route incoming requests toward, these techniques are the load balancer's routing algorithm.

Below lists several common routing techniques. There are many other generic and bespoke routing algorithms that are not listed.

Round Robin allocates requests to the next available server regardless of where the last request was sent. It is simple, and in practice, works effectively.

Least Connections sends the next request to the node with the fewest current connections. The load balancer is responsible for tracking how many connections exist to each node.

Weighted Least Connections sends the next request to the node with the least weighted connections. This is similar to the above least connections method, however, each node has an associated weight. This allows certain nodes to be preferred over others. This is useful if we have an inequal distribution of compute power. We would want to give smaller nodes a reduced load in comparison to other more powerful nodes.

Consistent Hashing In some cases we may want a user to consistently be routed to a specific node. This is useful for multiple transactions that need to be done in a consistent order or if the data is stored/-cached on the node. This can be done by hashing the information in the request payload or headers and then routing the request to the node that handles hashes in the range of the computed hash.

2.2 Health Checks

When load balancing, it is important to ensure that the nodes we route requests to are able to service the request. A good health check can save or break your service. Consider the two following examples from UQ's Information Technology Services (ITS):

Example 1 Early in my career, I, Evan Hughes, setup a multi-node Directory Server at UQ under the hostname of `ldap.uq.edu.au`. This server was a NoSQL database which implemented the LDAP protocol and supported `UQ Authenticate`, UQ's Single Sign-On service.

The service had a load balancer which checked that `port 389` is open and reachable. This worked well most of the time. However, the health check was too weak. When:

1. A data-center outage occured; and
2. The operating system running the service disappeared; but
3. The service was still running in memory.

The health check passed, but in reality, the service was talking to dead nodes, causing upstream services to have intermitent failures.

TODO: C4 Diagram of the architecture

Example 2 During the rollout of a new prompt for UQ Authenticate which required users to go to my.UQ to provide verified contact details - the Blackboard (learn.uq.edu.au) service went completly offline. The health check for Blackboard at the time completed a full authentication as a test user to ensure everything was functioning as expected. Once this user was enrolled into the new rollout, the health checks started reporting failures and within a matter of minutes the entire pool of nodes were shutdown. This health check was too broad and was not isolated enough to the service that it was checking.

TODO: C4 diagram of the architecture

A lot of services will provide a health check endpoint or a metrics endpoint which can help the engineer setup a proper level of health check. We want a health check that is specific enough for the service that it is checking but not so specific that it is too brittle. For the TaskOverflow application that we have been building so far, a reasonable health check would be that the health endpoint ensures the database is available and that the application is able to connect to it.

3 Load Balancers in AWS

3.1 Types of AWS Load Balancers

Not all load balancers are the same. Some load balancers inspect the transmitted packets to correctly route the packet. We will cover two load balancer types AWS provides:

Application Load Balancer is an OSI layer 7[1] load balancer which routes traffic based on the request's content. This is useful for services using HTTP, HTTPS, or any other supported protocol.

Network Load Balancer is an OSI layer 4[2] load balancer which routes traffic based on the source and destination IP addresses and ports. This is useful for services that are using TCP or UDP.

3.2 AWS Load Balancer Design

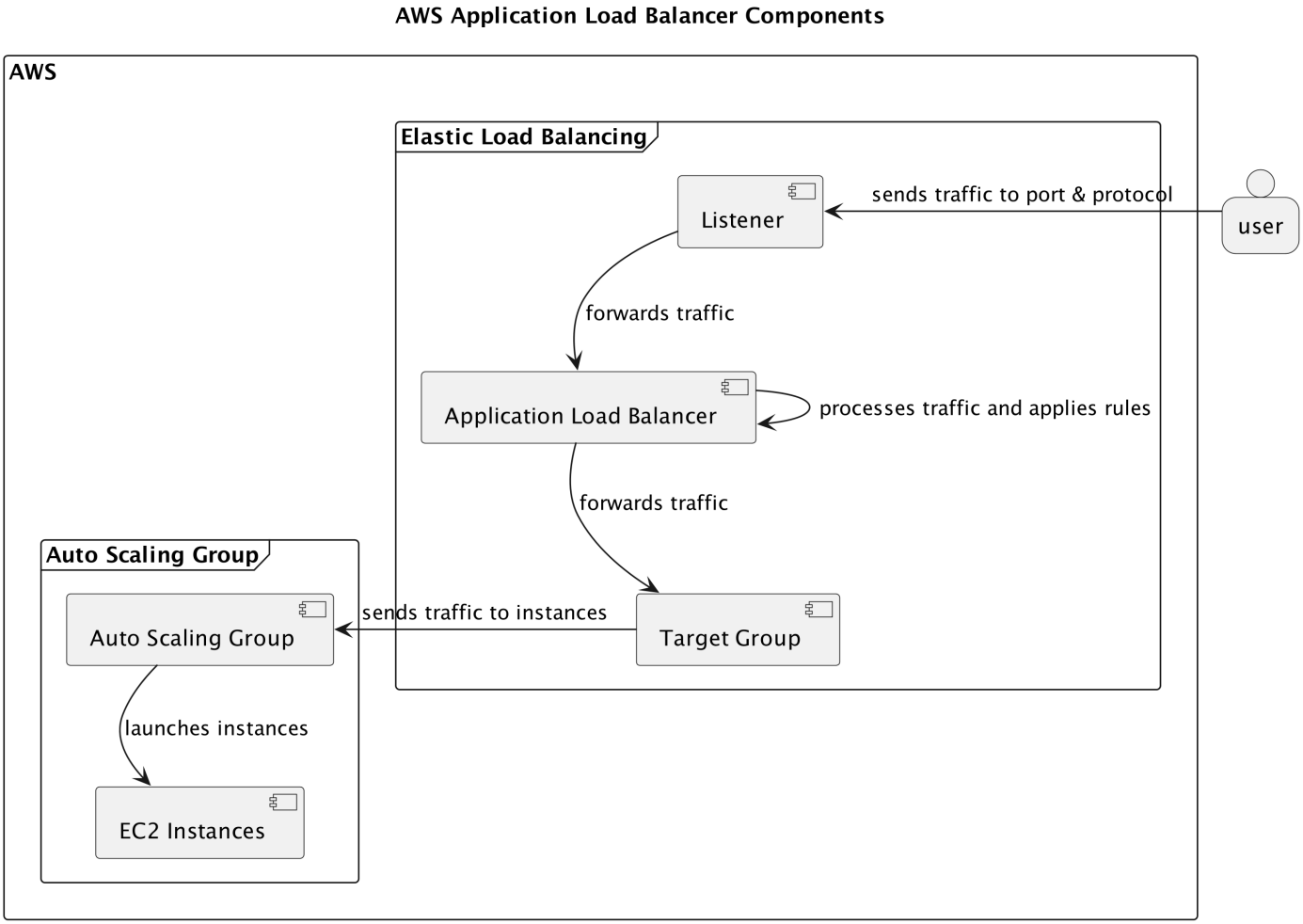
An AWS Elastic Load Balancer has three distinct components.

Listeners allows traffic to enter the Elastic Load Balancer. Each listener has a port (e.g. `port 80`) and a protocol (e.g. `HTTP`) associated with it.

Target Groups are groups of nodes which the load balancer can route to. Each target group has a protocol and a port associated with it, allowing us (the programmer) to switch ports on the way through the load balancer. This is useful if the targets are using a different port to the ports we want to expose.

Load Balancer is the actual load balancer that routes the traffic to the target groups based on rules that we setup. The load balancer has a DNS name that we can use to route traffic to it. The load balancer also has a security group that we can use to control what traffic can enter the load balancer.

[1] OSI layer 7: Application, in this case HTTP/HTTPS/etc
[2] OSI layer 4: Transport, in this case TCP/UDP



3.3 Autoscaling in AWS

Instead of creating the maxiumum amount of services we predict we will need, we can automatically scale the number of nodes we need to minimise resources. When the load is low, we operate with minimal nodes. When the load is high, we increase the number of nodes available to cope.

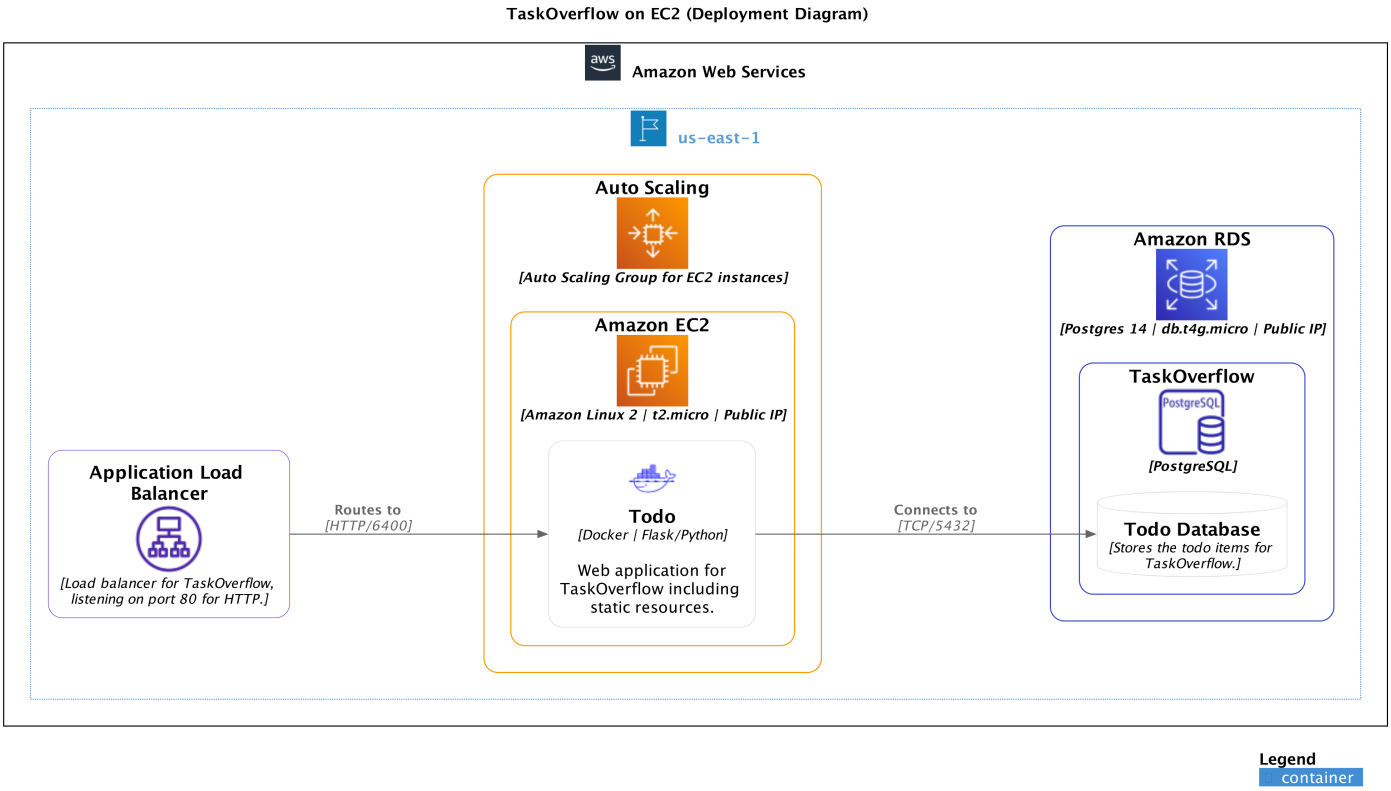
To compute the resources needed, AWS relies on triggers from CloudWatch and scaling policies. Some premade triggers are based around a node's;

- CPU usage,
- memory usage, or
- network usage.

We create custom triggers based on our application's metrics.

4 Load Balancing TaskOverflow

4.1 [Path A] EC2



TODO: Move to EC2Template

TODO: AutoScaling group

TODO: target group

TODO: autoscaling + target

TODO: load balancer

TODO: listener

TODO: Applying autoscaling rules

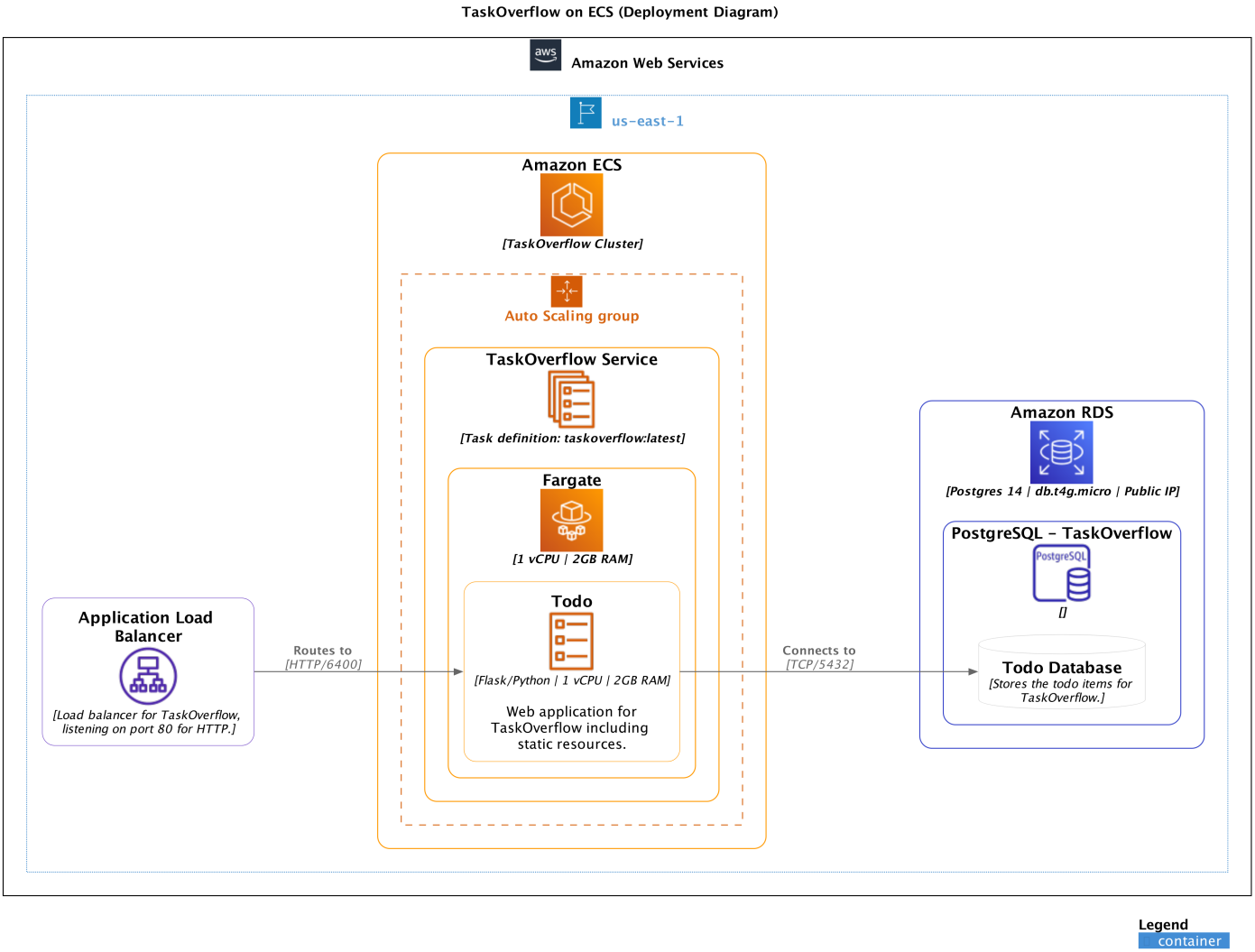
4.2 [Path B] ECS

TODO: load balancer

TODO: adding to ecs

TODO: listener

TODO: Applying autoscaling rules



4.3 Producing Load with K6

References