Live Captioning Kubernetes Readiness and Liveness Probes

Background Information

Kubernetes offers two types of probes - Readiness and Liveness. These are used to perform two very distinct functions:

- Readiness probes keep Kubernetes from sending requests to the pod when it is not ready to handle them. When a pod is started it is not considered ready until this probe has met its success criteria. This probe covers both situations where the application has to perform some level of processing on startup, as well as when the application may be overwhelmed with requests and unable to service any new requests until the existing requests have at least, in part, been satisfied. Failures that exceed the defined threshold cause Kubernetes to take that pod out of the list of pods that are available to handle requests.
- Liveness probes allow for the detection of an application that has become unresponsive. This can happen due to a variety of reasons. Failures that exceed the defined threshold cause Kubernetes to force a restart of the pod.

More information and background can be found in the Kubernetes documentation - Tasks section - Configure Liveness and Readiness Probes - link to the current level docs: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

Some additional articles on best practices, etc:

RedHat Openship Blog - Liveness and Readiness Probes: https://www.openshift.com/blog/liveness-and-readiness-probes

Kubernetes Readiness and Liveness Probes - Best Practices: https://medium.com/metrosystemsro/kubernetes-readiness-liveliness-probes-best-practices-86c3cd9f0b4a

From the above documentation and discussion some important points should be noted:

- Readiness probes should contain code that validates that the service and its immediate dependencies are available to handle requests. This means if a service is dependent upon a database connection for all of its requests, then that connection should be one of the items validated.
- Liveness probes should NOT contain code that invokes any external dependency. Rather a liveness probe should only prove that the main thread or HTTP server is able to respond. Optionally - this probe can respond with identifying information that might be helpful such as version information as well as some simple health information as long as that information is not expensive or memory intensive to obtain and return.

While Kubernetes offers several different forms of probes - Live Captioning will use the HTTP probes since that type is most suited for the various services provided.

Controlling Probe Timing and Success/Failure Thresholds

Kubernetes provides several tuning parameters shown in the table below that can be used to control how often the probes run, how long after the pod is started that Kubernetes will start the probe, how many consecutive failures it takes for the probe to consider the pod to have "failed", and how many consecutive successes are required before Kubernetes resets the accumulated failure count. See the table below for the specific setting name and explanation:

Setting	Default	Description
initialDelaySeconds	0	Number of seconds after the pod is started before Kubernetes will start running the probe
periodSeconds	10	Number of seconds between each probe attempt
timeoutSeconds	1	Number of seconds that Kubernetes will wait for a response before considering that probe attempt a failure
successThreshold	1	Number of successful results from the probe that is required to reset the failure count to 0
failureThreshold	3	Number of failures that are required before Kubernetes considers the pod to be in a failed state (see explanation above as "failed" means different things depending on whether this is a liveness or a readiness probe

See the design section below for the recommended changes from the defaults for these values.

Readiness Probe Design

- Each service will implement a readiness probe that will be available on the /check endpoint.
- If a service depends on another service for all requests, then it may use that service's /check endpoint to validate that the dependant service is also available. This provides consistency and ease of configuration for this probe across the range of services within WLC.
- If a service has a dependency that is not absolutely required then it should not be included in these checks. For example, even though Live Captioning depends on the Metering Service to record usage, if the metering service goes down - that does not cause a hard error for the Live Captioning service and thus should not be considered as part of the readiness checks within Live Captioning. L kewise - since Live Captioning

may use different ASR backend engines for any given request - and those are already defined in such a way as to eliminate a single point of failure for a given account (e.g. multiple Watson STT's endpoints, multiple AWS regions), those should also not be included in the code that determines the service's readiness.
- No checks should be made that require credentials that are not already known at service startup.

The following table contains the current list of services within WLC along with the checks (including any downstream services) that need to be made in that service's /check implementation.

Service	Checks
Authorization API	Database connection
Configuration	Database connection
Live Captioning	Authorization API /check Configuration /check Training API /check
Metering	Database connection
Metering Usage API	Metering /check
Rules API	Authorization API /check Database connection
STT Adapter	Training API /check
Text Processing	Components initialized?
Training API	Authorization API /check Database connection
Training Service	Database connection

Warning: With the above approach - you must avoid any kind of cyclic dependencies in these checks - otherwise you create a situation where you have a race condition on pod readiness and will prevent the pods involved from ever being declared ready.

Liveness Probe Design

- Each service will implement a liveness probe that will be available on the /health endpoint.
- Each service will respond with the following information:
    - Golang version (available from runtime.Version())
    - service version (available from main.version - set by Makefile to Git branch - tag values
    - Hostname (available from environment variable "HOSTNAME" if running in Kube which will give us the pod name, otherwise should be set to the hostname from os.Hostname())
    - Optional - Memory information such as HeapAlloc, HeapIdle and HeapInuse from the MemStats struct populated from a runtime. ReadMemStats() call (need to test to make sure this call is not going to cause any performance issues)

Timing Considerations

The basic thing we want to accomplish with adjusting the probe timing and threshold values is to ensure that in the case where the pods for a given service are flooded with requests - the pods are taken out of ready state soon enough and long enough to allow them to recover and continue processing the current requests such that the liveness probe does not end up firing and causing the pod to restart. This will help avoid interrupting and causing failures for those in-process requests. Below are the recommended adjustments from the defaults for these two probes. Absence of a value for a given probe - setting indicates that the default will be used.

Setting	Readiness	Liveness	Explanation
initialDelaySeconds		30	For liveness - there is no need to run this probe right away - and in fact - if the particular service has some expensive work to do at startup - it can actually cause repeated restarts for no reason.
periodSeconds			No change required for either.
timeoutSeconds	5	5	For readiness - this value should allow for readiness checks to complete - but fine tuning may be required depending on the service and the number of checks it needs to make. For liveness - this allows for a bit more leeway in responding when the pod is under heavy request load. General note - most of the services already had a timeout override set to 5 seconds - so this will just standardize this value for the few remaining services where this was not already in place.
succcessThreshold			No change required for either.
failureThreshold		5	For liveness - this is to give pods that are under duress due to high request rates time to be recover and start being able to respond to /health before being put back in service - thus avoiding unnecessary restarts.

With the above values - pods will be taken out of service after 3 successive failures of the /check endpoint - which would require 30 seconds. Pods will not be forced to restart unless there are 5 successive failures of the /health endpoint - which would require 50 seconds. That gives a pod that has been flooded with requests 20 seconds to recover and process enough to be able to satisfy the lightweight /health endpoint. If needed, we could adjust the periodSeconds value to further the difference between when the Readiness and Liveness probes would consider a pod to be in the "failed" state. However the above should give us a good starting point without having to adjust every value from their defaults.

If we need to adjust the above recommendations for a given service - the above table can be expanded to include those specific variations as needed.

Migration Considerations

Not all of the changes required to implement the above design will be done in one step. The initial implementation (covered in ticket WLC-1663) will handle implementing the above probes fully in the Live Captioning service, including all changes required to the Helm charts, etc. The other services will have the existing endpoint implementation for / swapped for both /check and /health until such time as tickets can be completed to implement the code in each of those services. Their Helm charts will also be updated as part of WLC-1663 so that only the code changes will need to be made for each in the future.