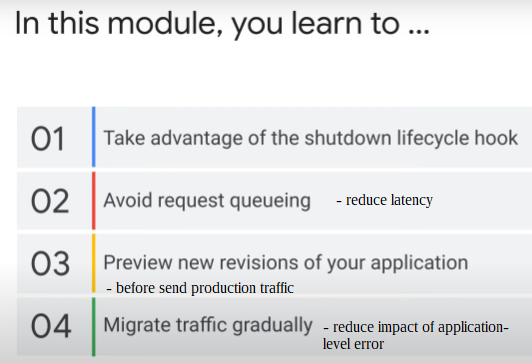
Application Development with Cloud Run

### Diving Deeper into Cloud Run

#### Introduction



### Container Lifecycle

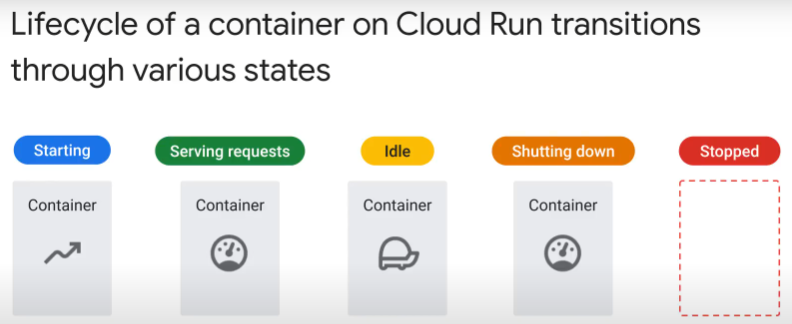
In its life cycle, a container transition through 5 states -

Start: begin application processes, based on container image

Serving requests: handling web requests

Idle: **not** handling web requests. \*\*important state\*\*

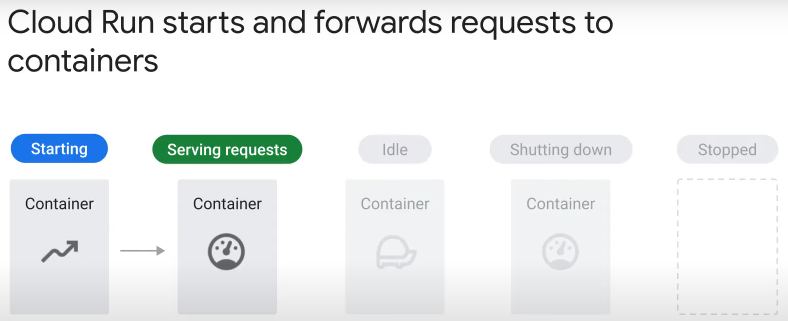
Shutting down: can shut down application gracefully if handle the shutdown hook.



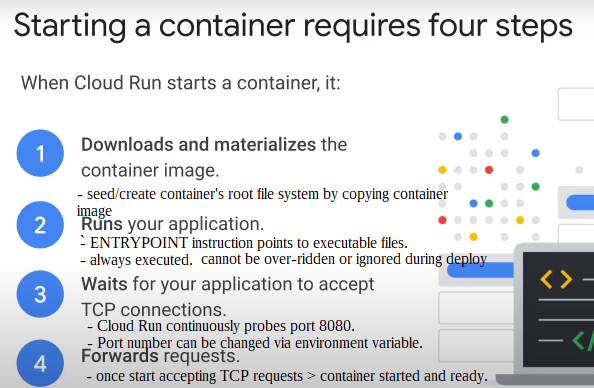
### Starting a container

Starting state

* begins when Cloud Run pulls container image
* ends when container starts serving web requests



4 steps needed to start a container:

* can do TCP-based readiness check for application. Listen to port in the code only when preceding code ready to handle requests.

Where does Cloud Run pull a container image from?

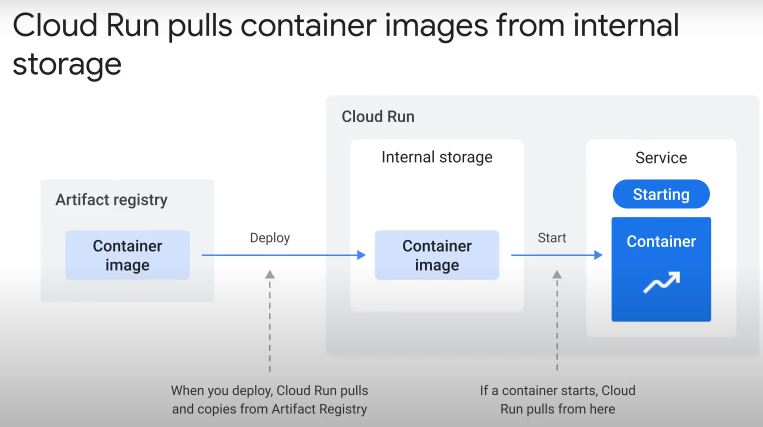
For 1st deployment (event) > from Artifact Registry

* Cloud Run then saves a copy to internal storage

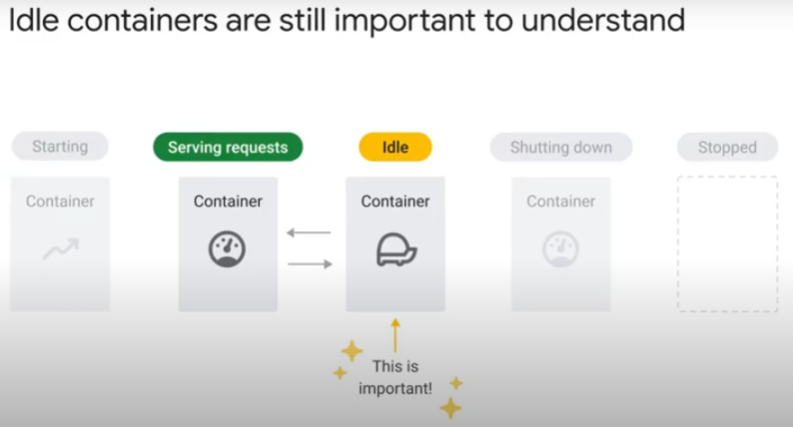
For 2nd and subsequent starting of container from same container image (event) > from internal storage

* internal storage optimised to ensure large images load as fast as tiny ones
* Cloud Run insulated from failures in Artifact Registry
* Cloud Run still works if deployed image deleted from Artifact Registry

If container image changed > new revision > Cloud Run will pull and copy new image from Artifact Registry during deployment

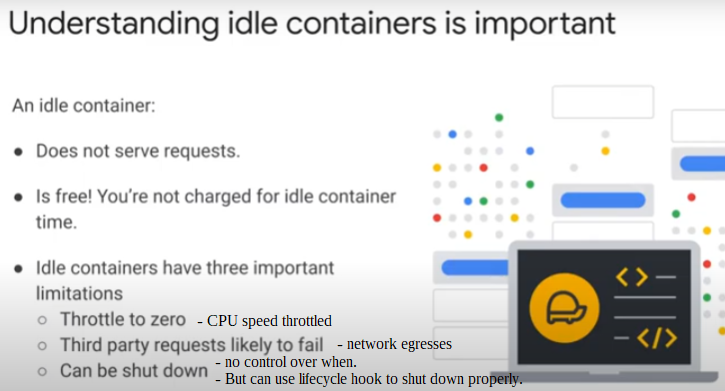


#### Serving Requests

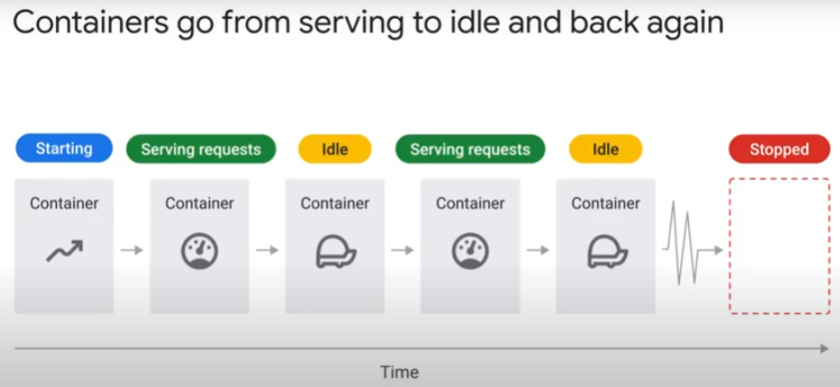


* A container is in the “serving requests” state if it is handling requests.
* If no requests handled for 100ms, container transitions into “idle” state.

Characteristics of an idle container:

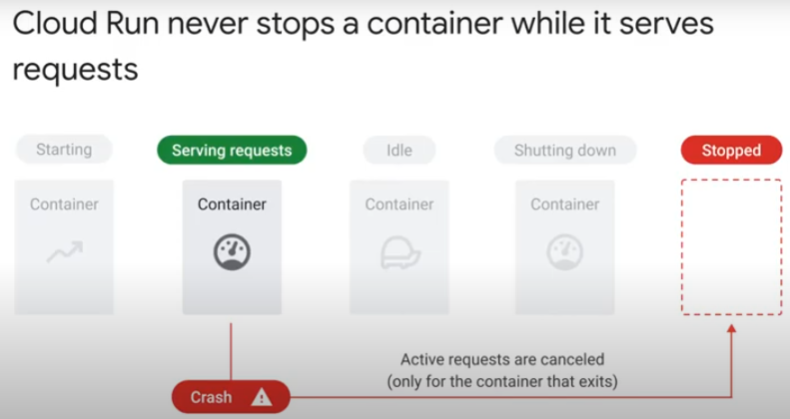


Container can transition betweem “serving requests” and “idle” many times:

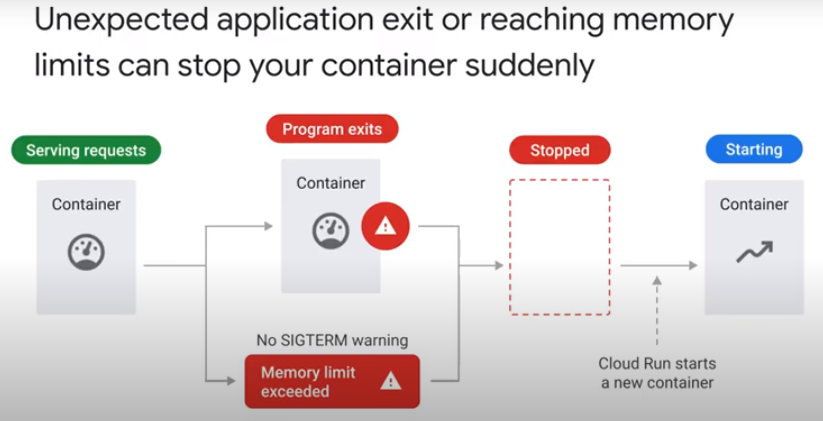


* when enter “serving requests” from “idle” state, Cloud Run service returns full access to CPU immediately.
  + User and application will not notice lag
* idling duration uncertain
  + an idle container can be stopped anytime
  + an idle container may be retained to serve traffic bursts (not guaranteed)
* can set minimum number of instances on Cloud Run
  + reserving idle containers. Prevent them from shutting down.
  + Faster response to requests (warm start, no cold start) than from stopped containers.

#### Handling Shutdown



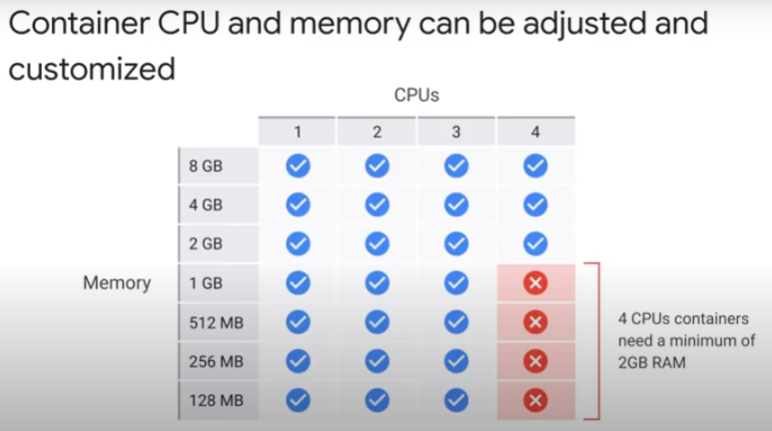
* Cloud Run never stops a normal container serving requests
* When container fails, **all** requests being handled by that container will fail.

A container instance is terminated (fails) for 2 reasons:

1. application exits, e.g. due to code failure
2. allowed memory limit of container exceeded

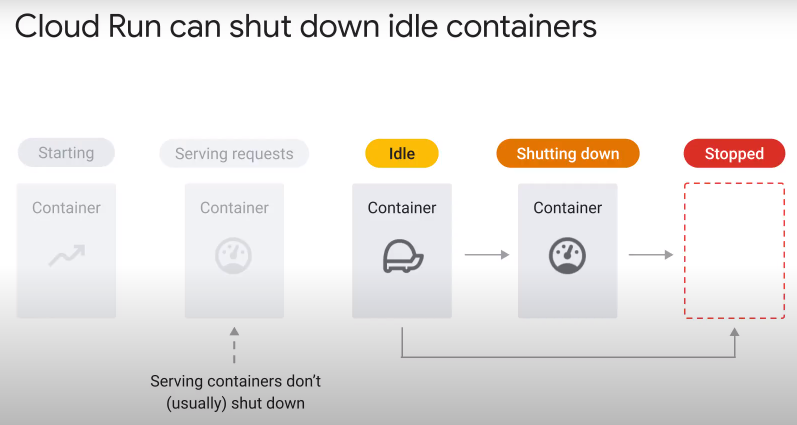
* default for each container job/revision is 512MB
* a soft limit – application may exceeds 512MB
* however, Cloud Run stops the container
* outputs a warning to the logs

Number of CPUs and memory limit for each container instance can be specified

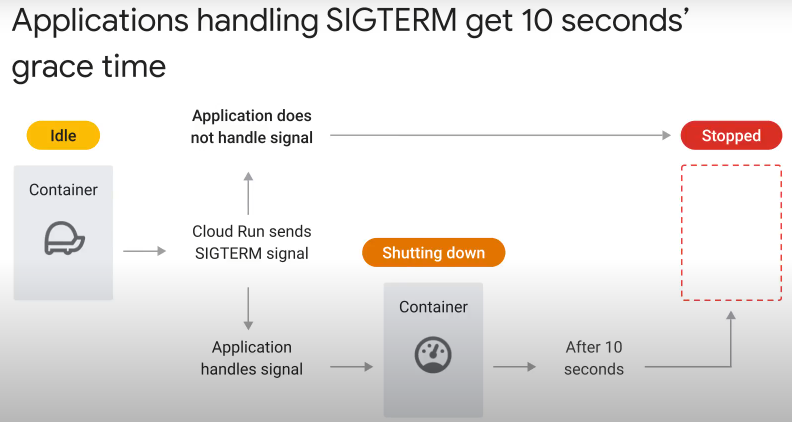


* up to 32GB memory (default is 512MB for gen2 execution environment)
* up to 8 CPUs (default is 1)
* any configuration change is a new revision, Subsequent revisions will get this config settings unless explicitly changed.
* Startup CPU boost – temporarily increase CPU allocation during startup to reduce startup latency
* https://cloud.google.com/run/docs/configuring/cpu
* 2 CPU allocation models
  + only during request processing, container startup and shutdown (default)
  + aloways allocated (still shutdown if idle >15min)
  + https://cloud.google.com/run/docs/configuring/cpu-allocation

Cloud Run shuts down / terminates idle containers

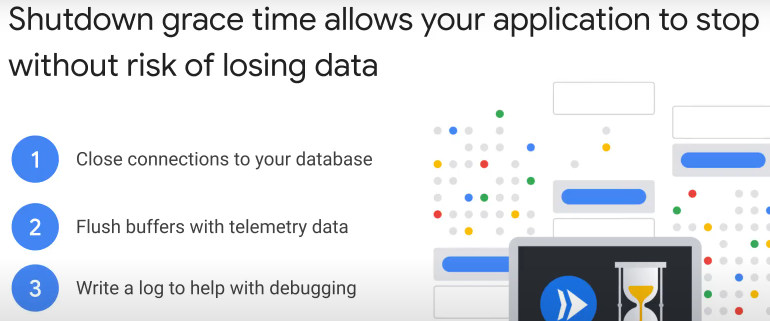
* cannot control when a shutdown happens to an idle container
* but can register a shutdown hook, so that container shuts down gracefully.

How SIGTERM works



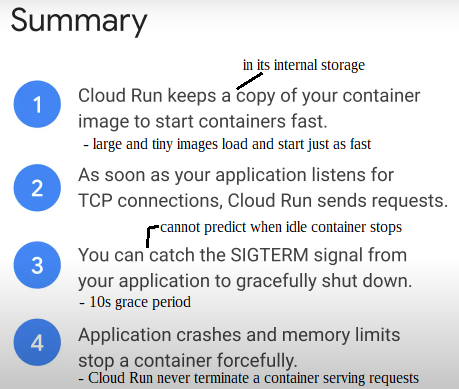
* Cloud Run sends idle container a SIGTERM (means TERMinate) signal
  + a Unix/Linux Kernel (OS) feature to terminate a process
  + the Unix/Linux process can handle, block or ignore SIGTERM signal. Cloud Run will output choice taken to logs.
* build your application to handle the SIGTERM signal
  + programming language may have library for hooking in to SIGTERM signal
  + initiate a clean orderly shutdown process
  + have 10s grace period
  + container has full access to CPU during 10s for graceful shutdown
  + at end of 10s, Cloud Run sends SIGKILL to terminate all processes immediately.
* if process does not handle SIGTERM signal, Cloud Run sends SIGKILL to terminate container immediately.

Examples of processes during shutdown



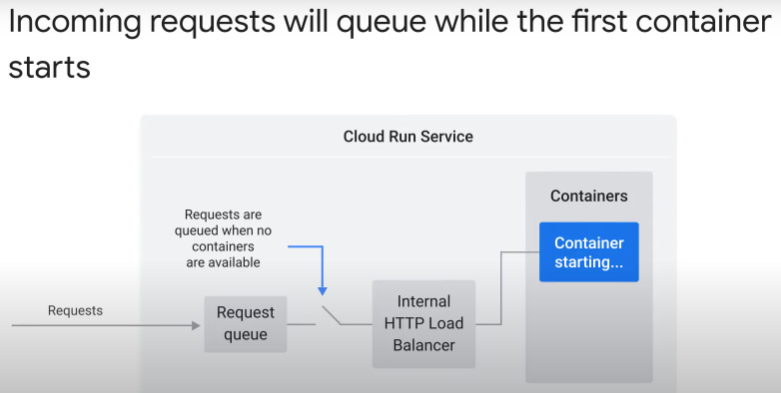
* complete transactions
* close network (e.g. TCP) connections.
  + downstream systems / databases may be slow to time-out idle connections.
  + May hit maximum connection allowed if don’t close idle connections
  + especially if Cloud Run is part of Kubernetes system with a lot of scaling
* save transient data to persistent storage
  + e.g. telemetry data from IoT devices, buffer data to be batched before sending to persistent storage.
* output to logs info that will help with debugging
* https://komodor.com/learn/sigterm-signal-15-exit-code-143-linux-graceful-termination/

Summary

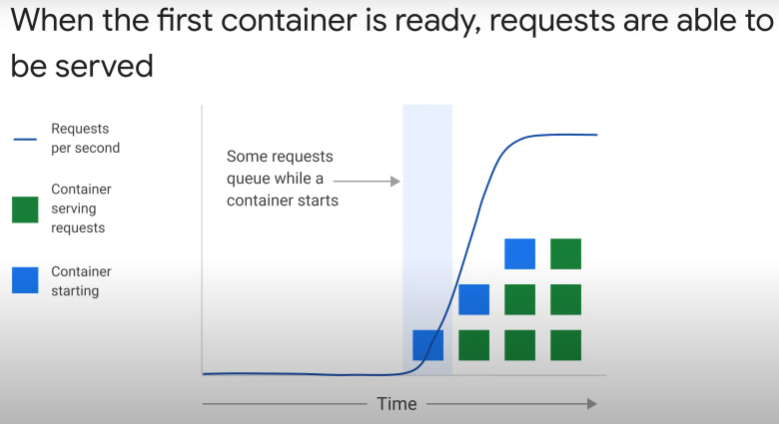


### Request queuing

* If Cloud Run does not receive requests for some time, it stops last idle container.
  + Scales down to zero container.
* Then, new requests coming in will queue while a new container starts
  + “cold start”
  + impact on latency of service

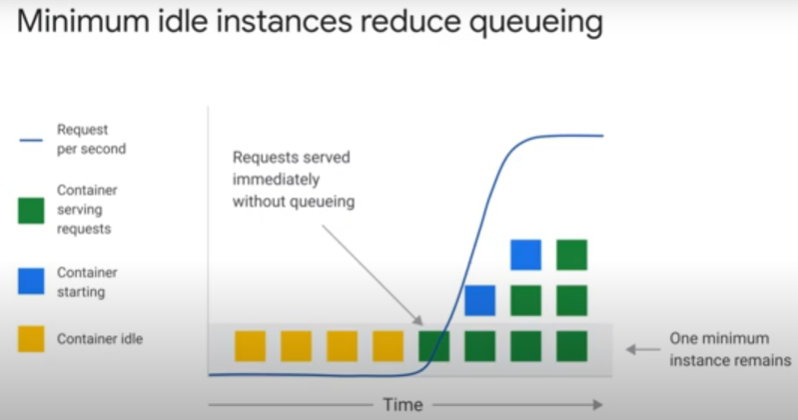


Incoming requests do not queue if there is a container available to handle requests



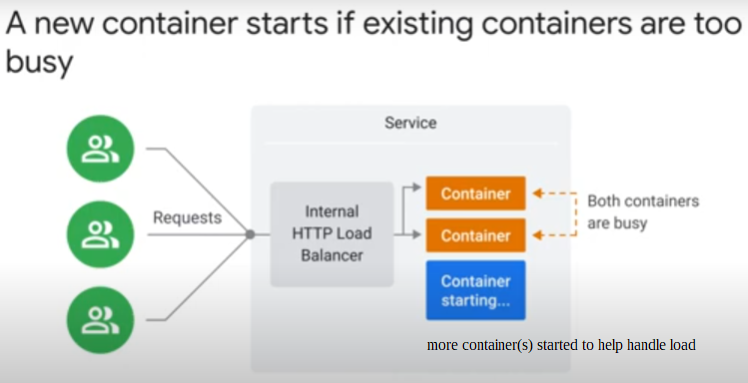
* blue vertical bar is startup latency
  + while first container starts
  + first few incoming requests will queue
  + downloading container image is very fast due to streaming technology and use of internal storage for Cloud Run.
  + Start container by running ENTRYPOINT command
  + wait for container to listen to configured port

Minimum idle instances reduce request queuing

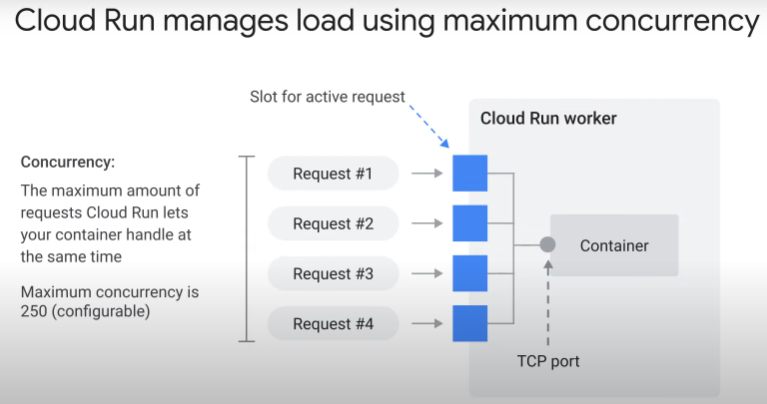


* can specify minimum container instances for Cloud Run
  + never remove last idle container instance
  + avoid cold start. No startup latency.
  + Will be charged for idle container but at reduced rate.

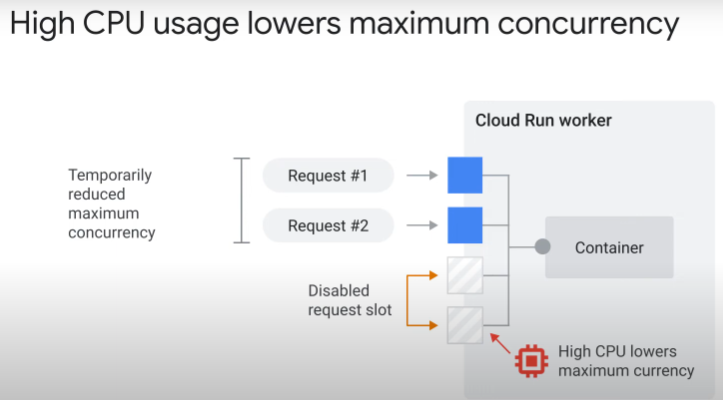
Auto-scaling to meet increased incoming traffic (load)



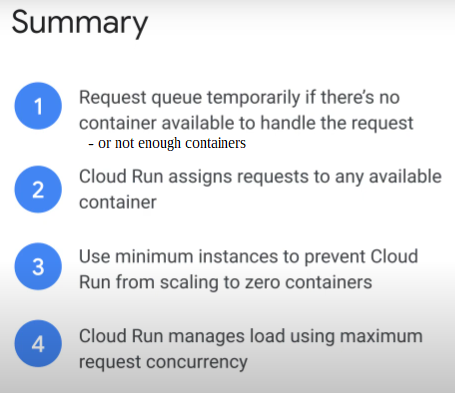
Maximize concurrency to handle load

* a Cloud Run instance can serve multiple requests simultaneously, up to a configurable max concurrency
  + this “concurrency” can be optimized and stress-tested
  + can be limited by technology stack, shared resources and code-level config.
  + [https://cloud.google.com/run/docs/tips/general#:~:text=While%20Cloud%20Run%20de%2Dcouples,called%20a%20%22cold%20start%22](https://cloud.google.com/run/docs/tips/general" \l ":~:text=While Cloud Run de-couples,called a cold start").
* request-based auto-scaling
* Cloud Run will try to route max number of requests concurrently to same container
* if all request slots filled out, Cloud Run does not route new request to that container until a slot is freed up
* limit concurrency to 1 if
  + using shared data;
  + do not want container terminated because memory limit exceeded. Every request must use predictable amount of memory.
  + a single request is using most of CPU or memory allocated to a container.
  + Code cannot process parallel requests. E.g. global states/variables that cannot be shared.
  + https://cloud.google.com/run/docs/about-concurrency

If CPU usage by container is high, max concurrency is reduced.



* Cloud Run will not load container with more requests if CPU utilization is high.
  + High CPU usage reduces max concurrency
  + a feedback loop
  + no such feedback loop for memory usage



Summary

1, Queuing requests allow new containers to be started, without losing requests.

Cloud Run keeps copy of conainer image in internal storage to start container fast.

Large and tiny container images load just as fast.

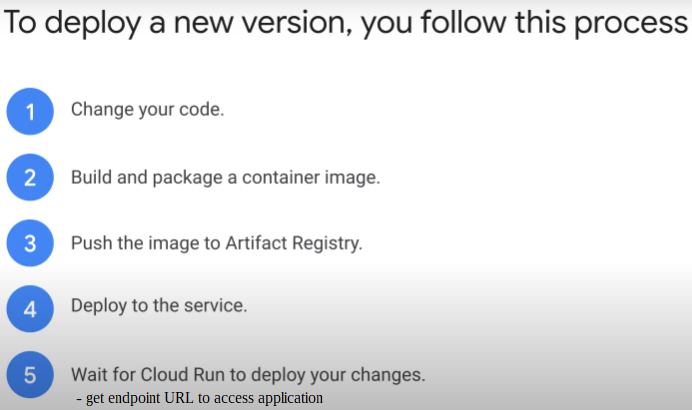
2. Container ready to handle web requests as soon as application listens for TCP connections.

3. Can have at least 1 container to handle influx of requests by configuring minimum instances in Cloud Run. Reduce latency of service compared to scale from zero.

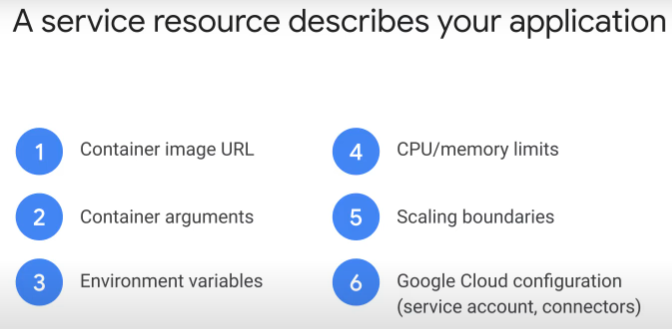
4. Can configure maximum concurrent requests per container instance. Default is 80.

#### Deployments

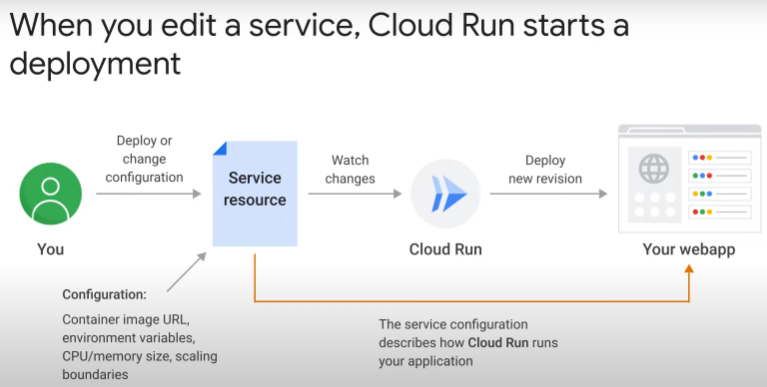
Steps to update an application deployed on Cloud Run:



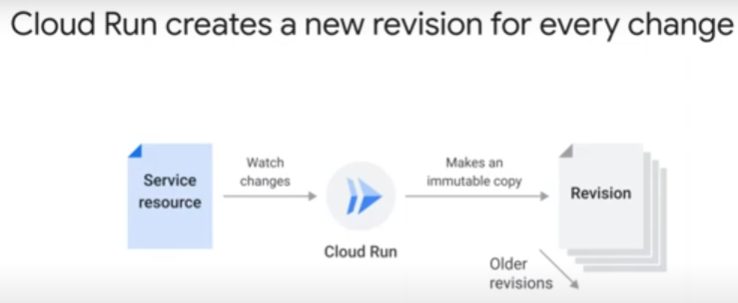
Service resource refers to the settings/configuration of Cloud Run:

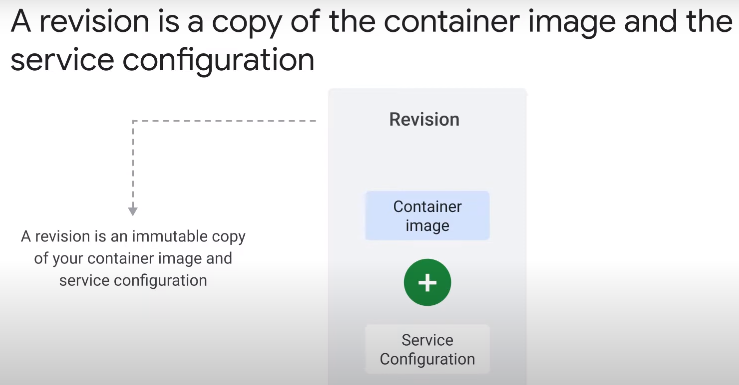


* Every change to a service resource parameter makes Cloud Run deploy a new revision of the application:



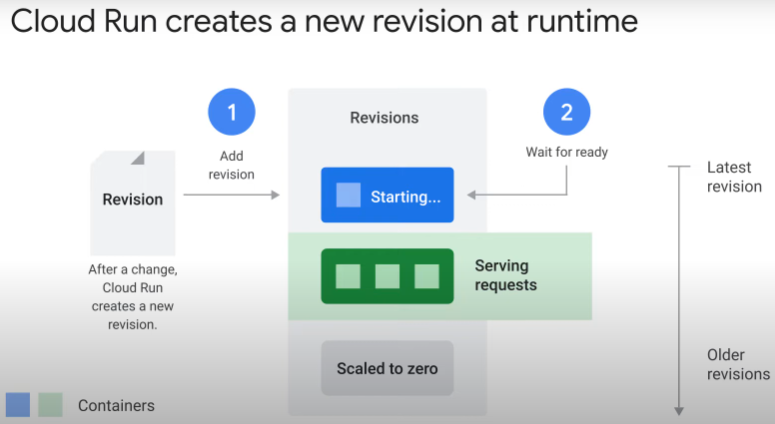
* a change to service resource -> Cloud Run detects change -> creates new revision -> re-deploy -> sends traffic to new revision the moment it is handling requests.

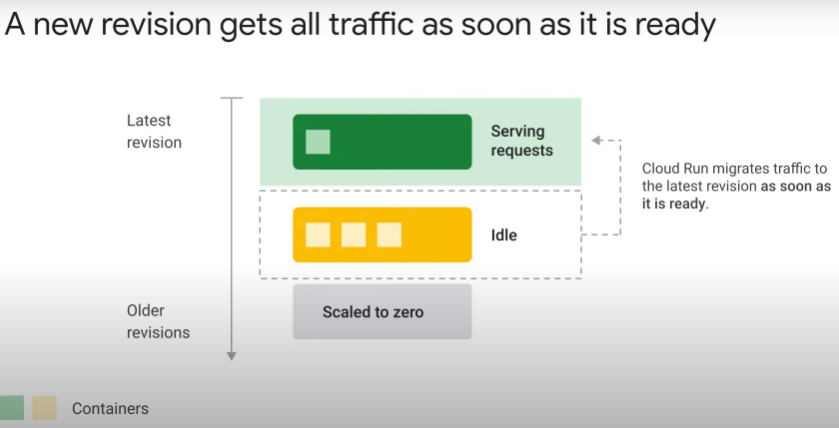


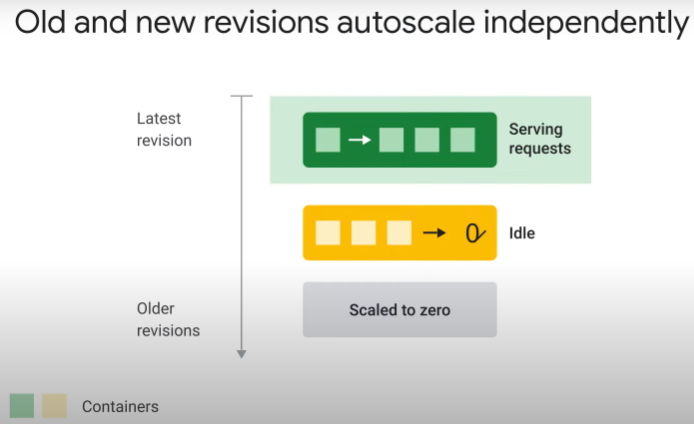


* A revision is an immutable copy of the service resource and container image.
  + Immutable means cannot be edited. Any update can only be saved as a new revision/version.

Cloud Run creates containers with new revision at runtime

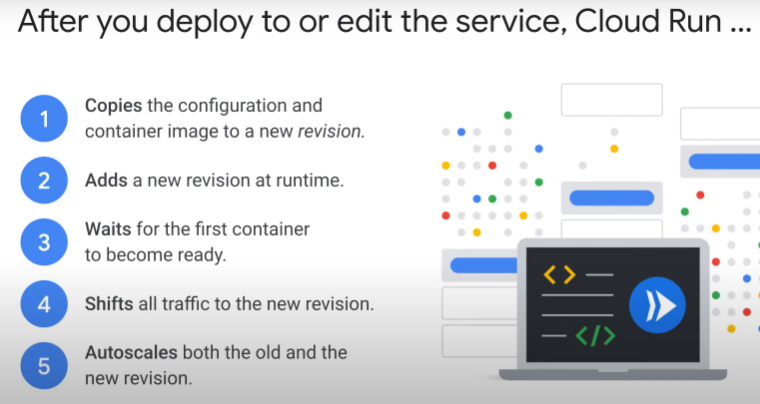


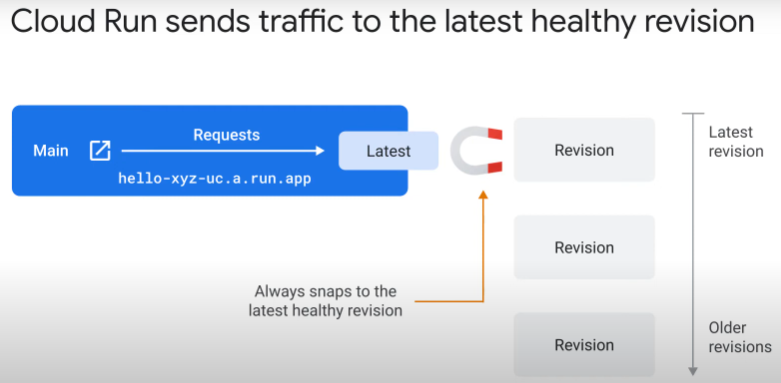
* Cloud Run deploys a new set of containers with latest revision at runtime.
  + waits for 1st container of new revision to start listening for TCP connections
  + containers of old revision still serve production traffic
* once 1st container of new revision ready to accept a TCP connection, Cloud Run sends all (100%) production traffic to it.
  + containers of old revision become idle, not serve requests
  + becomes status quo if no errors
  + note: Cloud Run does not look at response codes to determine error or not
  + if new revision is healthy, both old and new revisions scale independently
    - new: may scale up to handle load
    - old: scale down to zero



* this scaling behavior does not work well if application handles high traffic and takes time to start > will result in request queuing.
  + Solution: migrate traffic gradually

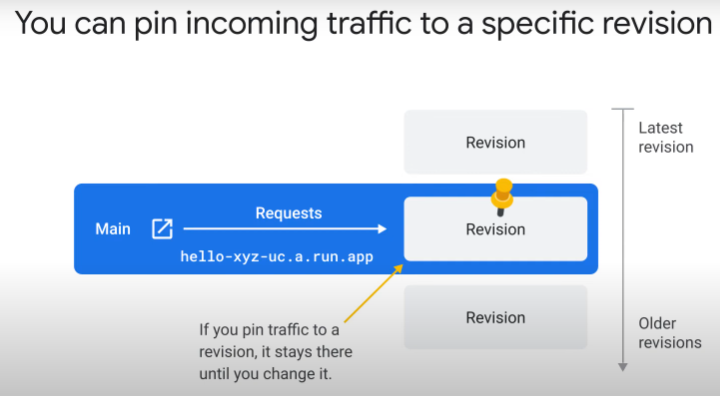
Summary of what happens after change to service resource:



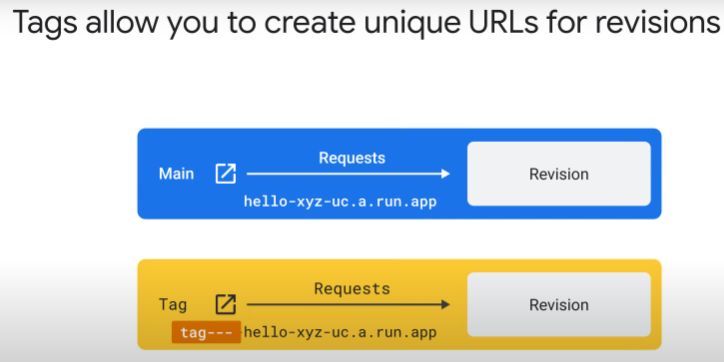


* Cloud Run snaps onto the latest ready revision
  + ready means able to serve HTTP requests

You can direct incoming traffic to a specific revision:

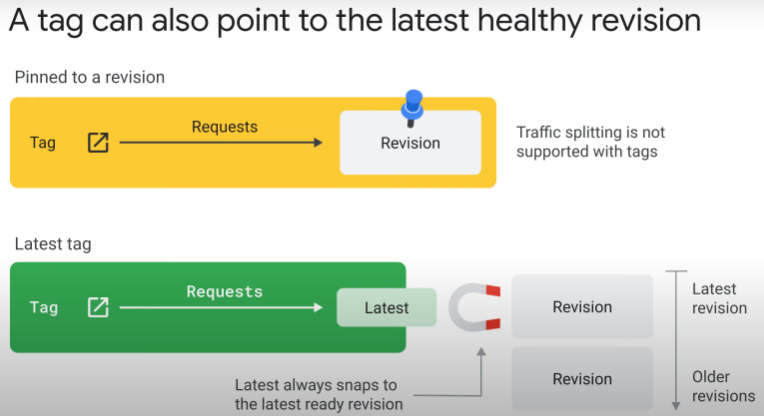
* If have new revision, Cloud Run will not automatically direct traffic to the latest revision.
* Useful when (1) rolling back deployment to previous revision; (2) need to test new revision before shifting all traffic to it.

You can tag a revision to create a unique service URL:



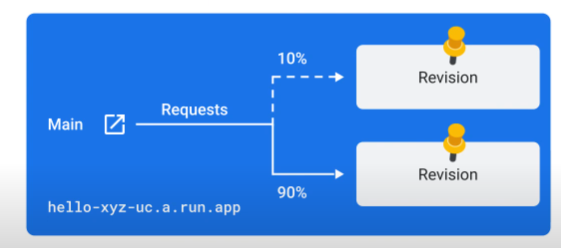
* tag added as prefix to service URL of Cloud Run. [tag]-xxx-run.app
  + e.g. automatically tag with ID of commit in version control that created the revision

A tag can be used to point to the latest healthy revision:



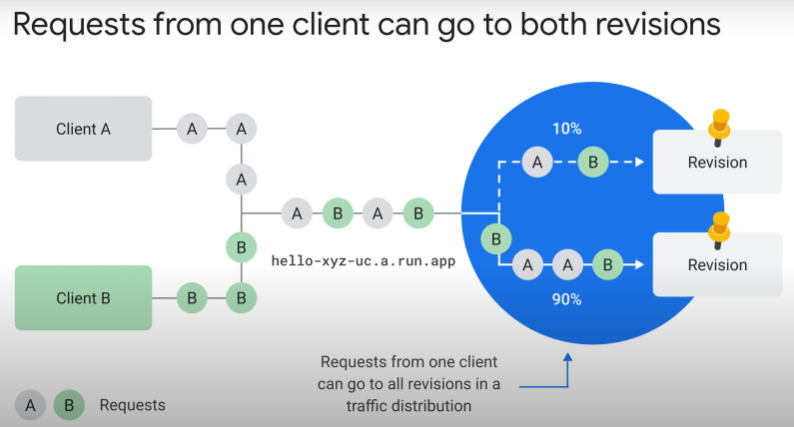
tags can be used for

* + preview of latest deployed revision
  + tests before migrating production traffic
    - run integration tests during development;
    - then, deploy container to staging environment ( no real traffic) and test against tagged revision in staging environ.
    - then, deploy to production environment and test against tagged revision (with no real traffic) in production environ.
    - finally, migrate production traffic gradually over days to tagged revision
      * can monitor for errors and scaling issues
      * not limited to 2 revisions, but most common way



* + direct traffic to particular revisions
* traffic splitting over multiple revisions not supported

Traffic split between multiple revisions:

* if you configure a traffic split, requests are divided between the revisions on a % basis
* one particular request can go to any of the revisions

Summary:

