**Requirements of Code Deployment System:**

We are building a system that involves repeatedly (in the order of thousands of times per day) building and deploying code to hundreds of thousands of machines spread out across 5-10 regions around the world.

Building code will involve grabbing snapshots of source code using commit SHA identifiers; beyond that, we can assume that the actual implementation details of the building action are taken care of. In other words, we don't need to worry about how we would build JavaScript code or C++ code; we just need to design the system that enables the repeated building of code.

Building code will take up to 15 minutes, it'll result in a binary file of up to 10GB, and we want to have the entire deployment process (building and deploying code to our target machines) take at most 30 minutes.

Each build will need a clear end-state (SUCCESS or FAILURE), and though we care about availability (2 to 3 nines), we don't need to optimize too much on this dimension.

Non-Functional Requirement:

How many PR or git commits or commits in main branch

Main branch – All the repo into consideration (Amazon) (3,00,000) 300k – (AWS, Amazon prime, amzon build, Amazon distribution, Amazon pay, Amazon heavy business, Amazon alexa, Amazon ware house Amazon fresh, Amazon Academy) (10 – 100-200 repositories

3 – 5 million builds commits (few auto commits, commit to branch and few main)

Stages – dev, test, stage and production

**Table Structure and SQL Queries:**  
  
We can have a jobs table in our SQL database where every record in the database represents a job, and we can use record-creation timestamps as the queue's ordering mechanism.

Our table will be:

id: string, the ID of the job, auto-generated

created\_at: timestamp

commit\_sha: string

name: string, the pointer to the job's eventual binary in blob storage

status: string, QUEUED, RUNNING, SUCCEEDED, FAILED

ACID transactions will make it safe for potentially hundreds of workers to grab jobs off the queue without unintentionally running the same job twice (we'll avoid race conditions). Our actual transaction will look like this:

How to make things syncronised:

BEGIN TRANSACTION;

SELECT \* FROM jobs\_table WHERE status = 'QUEUED' ORDER BY created\_at ASC LIMIT 1;

// if there's none, we ROLLBACK;

UPDATE jobs\_table SET status = 'RUNNING' WHERE id = id from previous query;

COMMIT;

All of the workers will be running this transaction every so often to dequeue the next job; let's say every 5 seconds. If we arbitrarily assume that we'll have 100 workers sharing the same queue, we'll have 100/5 = 20 reads per second, which is very easy to handle for a SQL database.

Failing Jobs to requeue:

UPDATE jobs\_table SET status = 'QUEUED' WHERE

status = 'RUNNING' AND

last\_heartbeat < NOW() - 10 minutes;

We can have a global service that continuously checks all regional GCS buckets and aggregates the replication status for successful builds (in other words, checks that a given binary in the main blob store has been replicated across all regions). Once a binary has been replicated across all regions, this service updates a separate SQL database with rows containing the name of a binary and a replication\_status. Once a binary has a "complete" replication\_status, it's officially deployable.