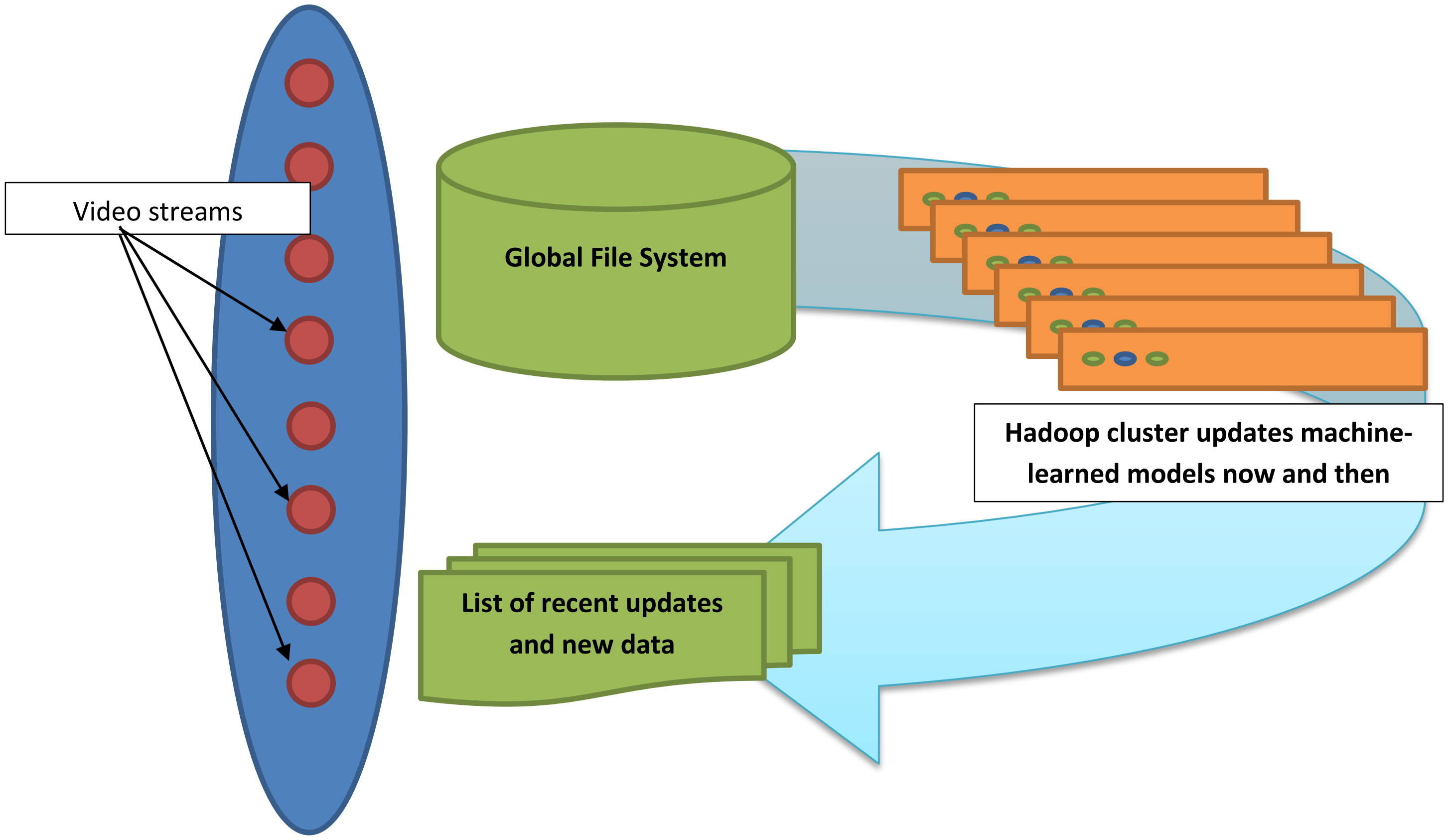
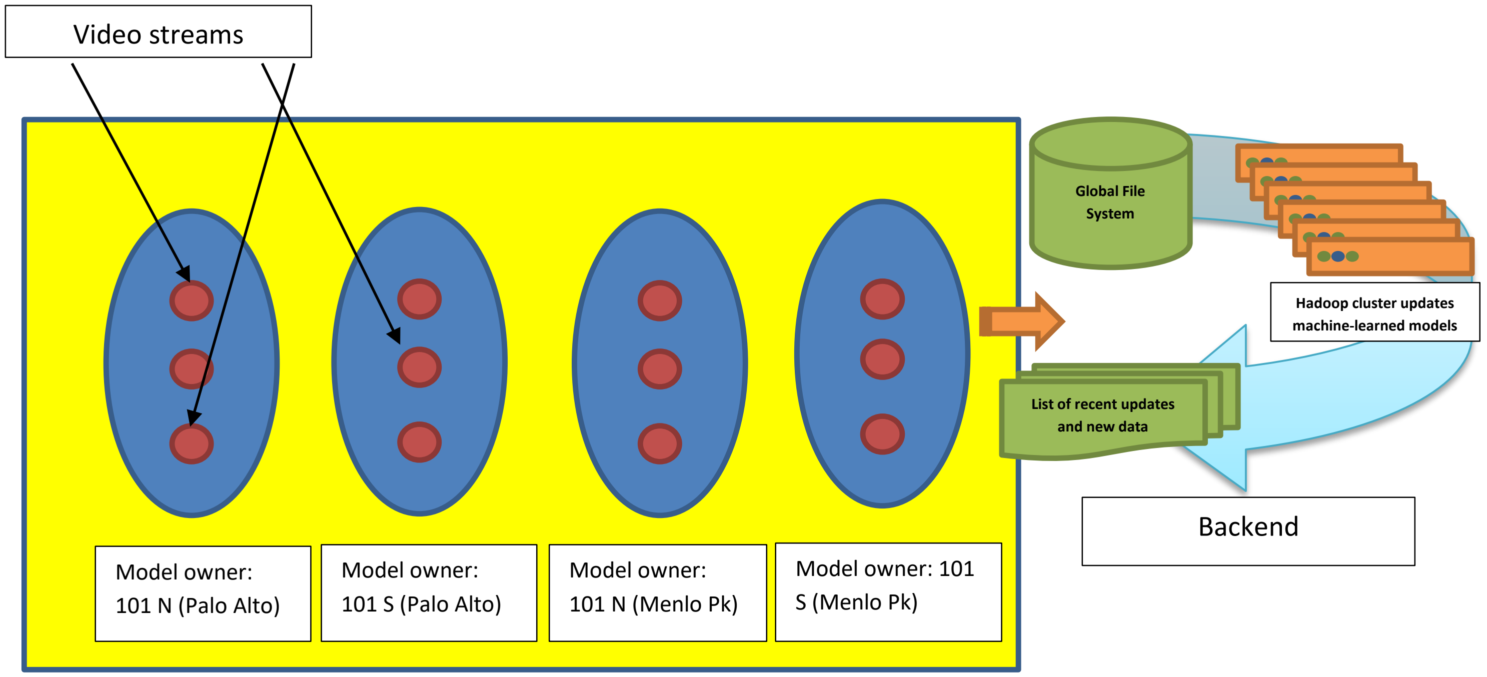
The classic way of building distributed and cloud computing systems (at least, over the past ten years) has focused on a “three tier” structure. The first tier systems are lightweight/stateless and just capture incoming requests, which they either handle locally (if read-only), or store into the global file system (if the request is an update). For the latter case, they also queue up a “new data to be processed” event. The second tier systems include the global file system and the cache. This would hold versions of data, such as machine learned models, on behalf of the first tier. This cached data might be stale (sometimes it takes a while for updates to reach it). The third tier systems run in the back-end and do heavy crunching. For example, with Hadoop/MapReduce, a machine learning system might wait for a batch of updates, then update machine-learned models a whole batch at a time, then distribute the updated models back into the second tier.


Video streams
Global File System
Hadoop cluster updates machine-learned models now and then
List of recent updates and new data

A new but uncommon option would be to move some part of the machine learning layer right into the data acquisition tier. So here, the first and second tier would merge, and would take over some roles that might historically have been handled “offline” in Hadoop. The argument for doing that is that it permits much faster reaction times (if it can be pulled off), but doing so also creates a need to use more powerful machines in the outer layer (the term “first tier” no longer would seem appropriate), and to replicate the machine-learned models, as well as the updates to them (we would move to an approach sometimes called “distributed” machine learning, in which a set of nodes independently propose adjustments to the model, and then the updates are somehow merged). We get a new picture:


Video streams
Global File System
Hadoop cluster updates machine-learned models
List of recent updates and new data
Backend
Model owner: 101 N (Palo Alto)
Model owner: 101 S (Palo Alto)
Model owner: 101 N (Menlo Pk)
Model owner: 101 S (Menlo Pk)

Think back to the smart highway scenario from last Thursday, but assume now that

- We are capturing video, not still images
- The density of cameras is enough to cover the entire highway in both directions, with high quality imaging and "K fault tolerance", meaning that even if K cameras fail, we still maintain adequate coverage. K might have a value like 2.
- The goal is to track vehicle trajectories, obtain a (driver,vehicle) image for each unique pair, but also to have a minimum of 30 seconds prior context stored for each vehicle (so that in the event of an accident, the behavior of that driver before the accident can be reviewed).

The question: which model does this argue for: the classic one, or the new one?

Which "capabilities" of the runtime environment are key to answering such a question? (Functionality available, performance, etc)?