Ensuring rollback safety during deployments

Sandeep Pokkunuri

One of the guiding tenets of how we build solutions at Amazon is: Avoid walking through one-way doors. It means that we stay away from choices that are hard to reverse or extend. We apply this tenet at all the steps of software development—from designing products, features, APIs, and backend systems to deployments. In this article, I'll describe how we apply this tenet to software deployments.

A deployment takes a software environment from one state (version) to another. The software may function perfectly well in either of those states. However, the software may not function well during or after the forward transition (upgrade or roll-forward) or backward transition (downgrade or rollback). When the software doesn't function well, it leads to a service disruption that makes it unreliable for customers. In this article, I assume that both versions of the software function as expected. My focus is on how to ensure that rolling forward or backward during deployment doesn't lead to errors.

Before releasing a new version of software, we test it in a beta or gamma test environment along multiple dimensions such as functionality, concurrency, performance, scale, and downstream failure handling. This testing helps us uncover any problems in the new version and fix them. However, it may not always be sufficient to ensure a successful deployment. We may encounter unexpected circumstances or suboptimal software behavior in production environments. At Amazon, we want to avoid putting ourselves in a situation where rolling back the deployment could cause errors for our customers. To avoid being in this situation, we fully prepare ourselves for a rollback before every deployment. A version of software that can be rolled back without errors or disruption to the functionality available in the previous version is called backwards compatible. We plan and verify that our software is backwards compatible at every revision.

Before I get into details about how Amazon approaches software updates, let's discuss some of the differences between standalone and distributed software deployments.

Standalone vs. distributed software deployments

For standalone software that runs as one process on one device, deployments are atomic. Two versions of the software never run at the same time. If the standalone software maintains state, then the new version must read (that is, deserialize) data written (that is, serialized) by the old version and vice versa. Satisfying this condition makes the deployment safe for rolling forward and backward.

In a distributed system, deployments become more complex. Deployments are done through rolling updates so that availability isn't affected. The new version is rolled out to a subset of hosts at once so that the other hosts can continue to serve requests. Typically, these hosts communicate with each other through a remote procedure call (RPC) or shared persistent state (for example, metadata or checkpoints). Such communication or shared state can present additional challenges. The writer and the reader could be running different versions of the software. As a result, they could interpret the data differently. The reader may even fail to read the data altogether, causing an outage.

Problems with protocol changes

We found that the most common reason for not being able to roll back is a change of protocol. For example, consider a code change that starts compressing data while persisting it to the disk. After the new version writes some compressed data, rolling back isn't an option. The old version doesn't know that it has to decompress data after reading from the disk. If the data is stored in a blob or a document store, then other servers will fail to read it even as the deployment is in progress. If this

data is passed between two processes or servers, then the receiver will fail to read it.

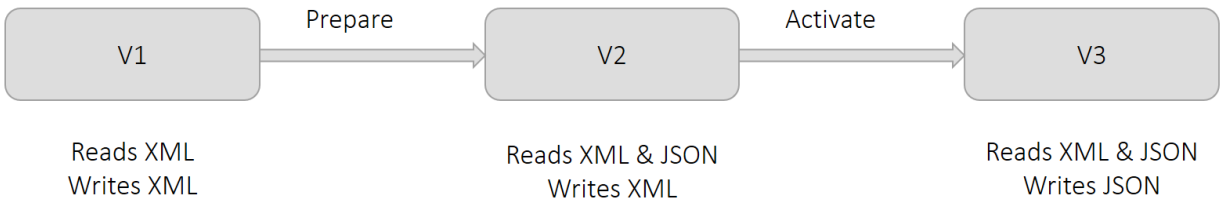
Sometimes, protocol changes can be very subtle. For example, consider two servers that communicate asynchronously over a connection. To keep each other aware that they are alive, they agree to send a heartbeat to each other every five seconds. If a server doesn't see a heartbeat within the stipulated time, it assumes that the other server is down and closes the connection.

Now, consider a deployment that increases the heartbeat period to 10 seconds. The code commit seems minor—just a number change. However, now it isn't safe to both roll forward and backward. During the deployment, the server running the new version sends a heartbeat every 10 seconds. Consequently, the server running the old version doesn't see a heartbeat for more than five seconds and terminates the connection with the server running the new version. In a large fleet, this situation can happen with several connections, leading to an availability drop.

Such subtle changes are hard to analyze by reading code or design documents. Therefore, we explicitly verify that each deployment is safe for rolling forward and backward.

Two-phase deployment technique

One way we ensure that we can safely roll back is by using a technique commonly referred to as two-phase deployment. Consider the following hypothetical scenario with a service that manages data (writes to, reads from) on Amazon Simple Storage Service (Amazon S3). The service runs on a fleet of servers across multiple Availability Zones for scaling and availability.
Currently, the service uses XML format to persist data. As shown in the following diagram in version V1, all the servers write and read XML. For business reasons, we want to persist data in JSON format. If we make this change in one deployment, servers that picked up the change will write in JSON. But, the other servers don't know how to read JSON yet. This situation causes errors. Therefore, we divide such a change into two parts and perform a two-phase deployment.



As shown in the preceding diagram, we call the first phase Prepare. In this phase, we prepare all the servers to read JSON (in addition to XML) but they continue to write XML by deploying version V2. This change doesn't alter anything from an operational standpoint. All the servers can still read XML, and all the data is still written in XML. If we decide to roll back this change, the servers will revert to a condition where they can't read JSON. This isn't a problem because none of the data has been written in JSON yet.

As shown in the preceding diagram, we call the second phase Activate. In this phase, we activate the servers to use JSON format for writing by deploying version V3. As each server picks up this change, it starts writing in JSON. The servers that have yet to pick up this change can still read JSON because they were prepared in the first phase. If we decide to roll back this change, all the data written by the servers that were temporarily in the Activate phase is in JSON. Data written by servers that were not in the Activate phase is in XML. This situation is fine because, as shown in V2, the servers can still

read both XML and JSON after the rollback.
Although the earlier diagram shows the serialization format change from XML to JSON, the general technique is applicable to all the situations described in the earlier Protocol changes section. For example, think back to the prior scenario in which the heartbeat period between servers had to be increased from five to 10 seconds. In the Prepare phase, we can make all servers relax the expected heartbeat period to 10 seconds although all the servers continue to send a heartbeat once every five seconds. In the Activate phase, we change the frequency to once every 10 seconds.

Precautions with two-phase deployments

Now, I will describe the precautions we take while following the two-phase deployment technique. Although I refer to the example scenario described in the earlier section, these precautions apply to most two-phase deployments.

Many deployment tools enable users to consider a deployment successful if a minimum number of hosts pick up the change and report themselves as healthy. For example, AWS CodeDeploy has a deployment configuration called minimumHealthyHosts.

A critical assumption in the example two-phase deployment is that, at the end of the first phase, all the servers have been upgraded to read XML and JSON. If one or more servers fail to upgrade during the first phase, then they would fail to read data during and after the second phase. Therefore, we explicitly verify that all the servers have picked up the change in the Prepare phase.

When I was working on Amazon DynamoDB, we decided to change the communication protocol between massive numbers of servers spanning multiple microservices. I coordinated the deployments between all the microservices so that all the servers reached the Prepare phase first and then proceeded to the Activate phase. As a precaution, I explicitly verified that the deployment succeeded on every single server at the end of each phase.

While each of the two phases is safe for rolling back, we can't roll back both changes. In the earlier example, at the end of the Activate phase, the servers write data in JSON. The software version in use before the Prepare and Activate changes doesn't know how to read JSON. Therefore, as a precaution, we let a considerable period of time pass between the Prepare and Activate phases. We call this time the bake period, and its duration is usually a few days. We wait to make sure that we don't have to roll back to an earlier version.

After the Activate phase, we can't safely remove the software's ability to read XML. It isn't safe to remove because all the data written before the Prepare phase is in XML. We can only remove its ability to read XML after ensuring that every single object has been rewritten in JSON. We call this process backfilling. It may require additional tooling that can run concurrently while the service is writing and reading data.

Best practices for serialization

Most software involves serializing data—whether for persistence or transfer over a network. As it evolves, it's common for the serialization logic to change. Changes can range from adding a new field to completely changing the format. Over the years, we have arrived at some best practices we follow for serialization:

- We generally avoid developing custom serialization formats.

  The initial logic for custom serialization may seem trivial and even provide better performance. However, subsequent iterations of the format present challenges that have already been solved by well-established frameworks such as JSON, Protocol Buffers, Cap'n Proto, and FlatBuffers. When used appropriately, these frameworks provide safety features such as escaping, backwards compatibility, and attribute existence tracking (that is, whether a field was explicitly set or implicitly assigned a default value).

- With each change, we explicitly assign a distinct version to serializers.

  We do this independent of source code or build versioning. We also store the serializer version with the serialized data or in the metadata. Older serializer versions continue to function in the new software. We find it's usually helpful to emit a metric for the version of data written or read. It provides operators with visibility and troubleshooting information if there are errors. All of this applies to RPC and API versions, too.
- We avoid serializing data structures that we can't control.

  For example, we could serialize Java's collection objects using reflection. But, when we try to upgrade the JDK, the underlying implementation of such classes may change, causing deserialization to fail. This risk also applies to classes from libraries shared across teams.
- Typically, we design serializers to allow the presence of unknown attributes.

  Where feasible, our serializers retain unknown attributes while writing back the data. With this accommodation, even if a server running the new version of software includes new attributes in the data while serializing, servers running the old version will not wipe out the attributes while updating the same data. Thus, a two-phase deployment isn't necessary.

As with many of our best practices, we share them with the caution that our guidelines are not applicable for all applications and scenarios.

Verifying that a change is safe for rollback

Generally, we explicitly verify that a software change is safe to roll forward and backward through what we call upgrade-downgrade testing. For this process, we set up a test environment that is representative of production environments. Over the years, we have identified a few patterns we avoid when setting up test environments.

I have seen situations where deploying a change in production caused errors although the change had passed all the tests in the test environment. On one occasion, services in the test environment had only one server each. Thus, all deployments were atomic which precluded the possibility of running different versions of the software concurrently. Now, even if test environments don't see as much traffic as production environments, we use multiple servers from different Availability Zones behind each service, just as it would be in production. We love frugality at Amazon, but not when it comes to ensuring quality.

On another occasion, the test environment had multiple servers. However, the deployment was done to all the servers at once to accelerate testing. This approach also prevented the old and new versions

of the software from running at the same time. The problem with roll-forward was not detected. We now use the same deployment configuration in all test and production environments.

For changes that involve coordination between microservices, we maintain the same order of deployments across microservices in test and production environments. However, the order for rolling forward and backward could be different. For example, we generally follow a specific order in the context of serialization. That is, readers go before writers while rolling forward whereas writers go before readers while rolling backward. Appropriate order is commonly followed in test and production environments.

When a test environment setup is similar to the production environments, we simulate the production traffic as closely as possible. For example, we create and read several records (or messages) in quick succession. All the APIs are exercised continuously. Then, we take the environment through three stages, each of which lasts for a reasonable duration to identify potential bugs. The duration is long enough for all the APIs, backend workflows, and batch jobs to run at least once.
First, we deploy the change to about half of the fleet to ensure software version coexistence. Second, we complete the deployment. Third, we initiate the rollback deployment and follow the same steps until all servers run the old software. If there are no errors or unexpected behavior during these stages, then we consider the test successful.

Conclusion

Ensuring that we can roll back a deployment without any disruption for our customers is critical in making a service reliable. Explicitly testing for rollback safety eliminates the need to rely on manual analysis, which can be error-prone. When we discover that a change isn't safe for rolling back, typically we can divide it into two changes, each of which is safe to roll forward and backward.