At Amazon, we often see patterns in our services in which a complex operation is decomposed into a controlling process making calls to a number of smaller services, each responsible for one part of the overall workflow. For example, consider the launch of an Amazon Elastic Compute Cloud (EC2) instance. "Under the hood" this involves calls to services responsible for making placement decisions, creating Amazon Elastic Block Store (EBS) volumes, creating elastic network Interfaces, and provisioning a virtual machine (VM). On occasion, one or more of these service calls might fail. To deliver a good customer experience we want to drive this workflow to success. To achieve this, we look to the controlling process to shepherd all decomposed services into a known good state.

We've found that in many cases the simplest solution is the best solution. In the scenario I just described, it would be best to simply retry these calls until they succeed. Interestingly, as Marc Brooker explains in the article Timeouts, retries, and backoff with jitter, a surprisingly large number of transient or random faults can be overcome by simply retrying the call. As simple as it seems, this pattern has proven so effective that we have baked default retry behavior into some of our AWS SDK implementations. These implementations automatically retry requests that fail as a result of network IO issues, server-side fault, or service rate limiting. Being able to simply retry requests reduces the number of edge cases that need to be dealt with by the client. As a result, it reduces the amount of undifferentiated boilerplate code needed in calling services. Undifferentiated boilerplate code in this context refers to the code needed to wrap service calls to remote services to handle various fault scenarios that might arise.

However, retrying a service call as mitigation for a transient fault is based on a simplifying assumption that an operation can be retried without any side effects. Put another way, we want to make sure that the result of the call happens only once, even if we need to make that call multiple times as part of our retry loop. Going back to our earlier example, it would be undesirable for the EC2 instance launch workflow to retry a failed call to create an EBS volume and end up with two EBS volumes. In this article, we discuss how AWS leverages idempotent API operations to mitigate some of the potential undesirable side effects of retrying in order to deliver a more robust service while still leveraging the benefits of retries to simplifying client-side code.

Retrying and the potential for undesirable side effects

To dive into this more deeply let's consider a hypothetical scenario where a customer is using the Amazon EC2 RunInstances API operation. In our scenario, our customer wants to run a *singleton* workload, which is a workload that requires "at most one" EC2 instance running at any time. To achieve this, our customer's provisioning process asks Amazon EC2 to launch this new workload. However, for some reason, perhaps due to a network timeout, the provisioning process receives no response.

The previous diagram shows the preparation of a semantically equivalent response in cases where the request has already been seen. It could be argued that this is not required to meet the letter of the law for an operation to be idempotent. Consider the case where a hypothetical CreateResource operation is called with the unique request identifier123. If the first request is received and processed but the response never makes it back to the caller then the caller will retry the request with identifier 123. However, the resource might now have been created as part of the initial request. One possible response to this request is to return a ResourceAlreadyExists return code. This meets the basic tenets for idempotency because there is no side effect for retrying the call. However, this leads to uncertainty from the perspective of the caller because it's not clear whether the resource was created as a result of this request or the resource was created as the result of an earlier request. It also makes introducing retry as the default behavior a little more challenging. This is because, although the request had no side effects, the subsequent retry and resultant return code will likely change the flow of execution for the caller. Now the caller needs to deal with the resource already existing even in cases where (from their perspective) it did not exist before they made the call. In this scenario, although there is no side effect from the service perspective, returning ResourceAlreadyExists has a side effect from the client's perspective.

Semantic equivalence and support for default retry strategies

An alternative is to deliver a semantically equivalent response in every case for the same unique request identifier for some interval. This means that any subsequent response to a retry request from the same caller with the same unique client request identifier will have the same meaning as the first response returned for the first successful request. This approach has some really useful properties—especially where we want to improve the customer experience by safely and simply retrying operations that experience server-side faults, just as we do with the AWS SDK through default retry policies.

We can see an example of idempotency with semantically equivalent responses and automated retry logic in action when we use the Amazon EC2 RunInstances API operation and the AWS Command Line Interface (CLI). Note that the AWS CLI (like the AWS SDK) supports a default retry policy, which we are using here. In this example, we launch an EC2 instance using the following AWS CLI command:

```
$ aws ec2 run-instances --image-id ami-04fcd96153cb57194 --instance-type t2.micro
{
    "Instances": [
        {
            "Monitoring": {
                "State": "disabled"
            },
            "StateReason": {
                "Message": "pending",
                "Code": "pending"
            },
            "State": {
```