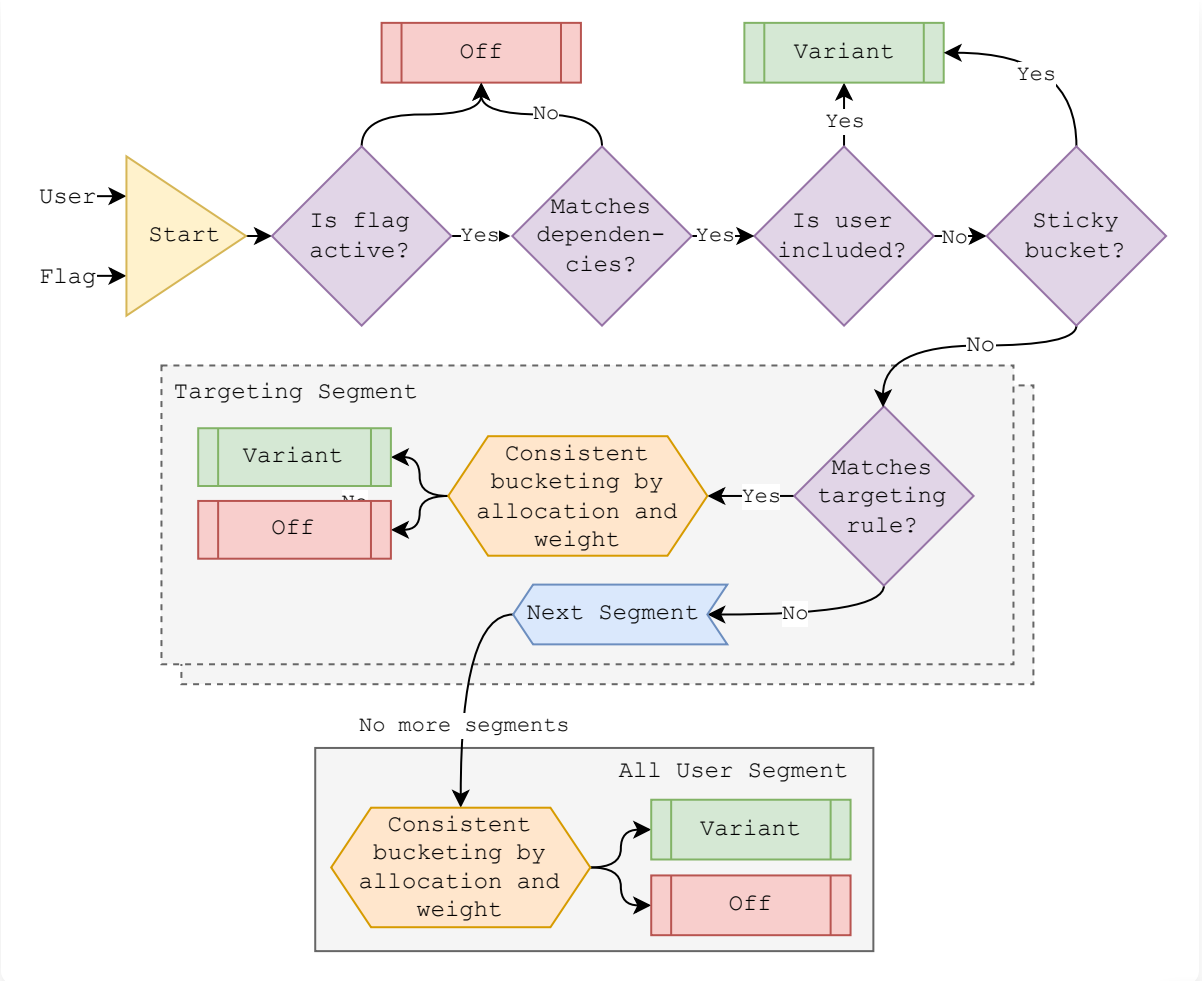
Implementation

Copy page

Evaluation refers to the act of determining which variant, if any, a user is bucketed into given a flag configuration. In short, evaluation is a function of a user and a flag configuration which outputs a variant.



Ask AI

Activation

A flag may be active or inactive. Inactive flags never return a variant as a result of evaluation.

Best practice

For simple on/off flags, Amplitude recommends using the all users segment allocation set to either 100% or 0% rather than using the activation toggle to control traffic. The activation toggle should be used to sunset a feature that has been fully rolled out or rolled back after the flag's instrumentation has been removed.

Flag dependencies

A flag may define a dependency on another flag's evaluation. If the dependency isn't met then no variant returns, otherwise the evaluation continues. Flag dependencies are currently utilized to implement mutual exclusion groups and holdout groups.

Example

For example, Flag-2 may define a dependency on Flag-1 evaluating to the variant `on`.

- Flag-1 (50% `on`)
- Flag-2 (50% `control`, 50% `treatment`)
  - Depends on Flag-1= `on`

The dependency ensures that Flag-1 will always be evaluated before Flag-2. Further, if Flag-1 evaluates to `on`, then Flag-2 will be fully evaluated. If Flag-1 does not evaluate to a variant, or to a variant other than `on`, the evaluation of Flag-2 fails the dependency check and no variant is assigned.

In this example, 50% of evaluated users will be assigned a variant for Flag-2.

Individual inclusions

Inclusions allow you to force bucket specific users (identified by either their user ID or device ID) into a variant. This feature is primarily used for development purposes. For example, if you are

Sticky bucketing

Warning

Sticky bucketing should be used with care. Even if sticky bucketing is disabled, consistent bucketing means that users are still bucketed into the same variant given that the user and targeting rules remain static. Changing targeting rules on an active flag with sticky bucketing enabled may cause a sample ratio mismatch (SRM), which may skew experiment results.

If sticky bucketing is enabled, a user will always get evaluated to the same previously bucketed variant, regardless of the current targeting. Sticky bucketing doesn't apply if the user hasn't been bucketed into a variant.

Targeting segments

Warning

Adding a target segment without defining any rules (where clauses) will capture all users even though the estimates show 0 users.

A flag or experiment may have `0-n` targeting segments. Targeting segments are evaluated from top-to-bottom. If a user matches the segment targeting rule, then consistent bucketing based on the configured allocation percentage and variant distribution weights determines which variant, if any, the user is bucketed into.

All users segment

The all users segment captures all users who don't match a targeting segment (if any are added). Users are bucketed into a variant (or no variant) via consistent bucketing based on the configured allocation percentage and variant distribution weights.

allocation percentage, and variant weights. In other words, given the same inputs, the output remains constant.

Input	Description
Bucketing Key	The key which determines which user property value to use as the bucketing value. The bucketing value is what's actually used as input to the hashing function.
Bucketing Salt	A string which is concatenated to the bucketing value before hashing. The bucketing salt is randomly generated when the flag or experiment is created and used indefinitely unless explicitly updated.
Allocation	The percentage of all users included in the segment who should receive a variant. Used in the allocation bucketing step.
Variant Weights	A weight given for each variant. Applied only to the percentage included by the allocation percentage. Used in the variant bucketing step.

The bucketing logic is split into two steps. The first step, allocation bucketing, determines if the user should be allocated a variant based on the allocation percentage. The second step, variant bucketing runs only if the user has been allocated in step one. Both steps use the same consistent hash function in slightly different ways.

The bucketing salt enables experiment allocation to be statistically independent. Without it, if Amplitude allocates a user to the treatment, they'd get the treatment in every experiment.

There are two cases in which you may need to update the bucketing salt:

1. You want to re-randomize users because of a bug or other issue in your experiment. In this case, update the salt to a new random string.
2. You want the evaluation of two experiments to be the same. In this case, update the salt to be the same in both projects.

Hashing

variant jump.

```
text
murmur3_x86_32("bucketing_salt/bucketing_value")
```

torchlight.dev

Allocation bucketing

A user is determined to be allocated if the hash value modulo 100 is less than the allocation configured in the segment.

```
text
murmur3_x86_32("bucketing_salt/bucketing_value") % 100
```

torchlight.dev

Variant bucketing

After a user is allocated, variant bucketing determines which variant the user should receive. Variants are associated with values between 0 and 42949672, based on their weights.

```
text
floor(murmur3_x86_32("bucketing_salt/bucketing_value") / 100)
```

torchlight.dev

For example, if variant `A` has weight 1, and variant `B` has weight 1, `A` would be associated with values in the interval `[0, 21474835]`, and variant `B` would be associated with values in the interval `[21474836, 42949672]`.

Amplitude

Get Started | Data | Analytics | Session Replay | Guides and Surveys | Admin | Partner

Terms of Service | Privacy Notice | Acceptable Use Policy | Legal