# **Feature Flag Framework User Manual**

| **Application Name** | **Feature Flag Framework** |
| --- | --- |
| **Version date** | 01/26/2025 |
| **On this page** | * 💡Best Practices   + 1. Feature Flag Management   + 2. Feature Flag Service   + 3. Feature Flag Client Library * ⏸️ Limitation |

# **Best Practices**

## **1. Feature Flag Management**

· Always register the new feature flag first using Feature Flag Management, before using Feature Flag Client: using Prod DAG. Feature Flag Client always pulls Feature Flag data from Prod Bigtable instance.  
If the feature flag name already exists, DAG will throw an error.  
Registration takes two steps:  
1. Add a new feature flag: one DAG run.  
2. Add a new version for this new feature flag: one DAG run.  
Details are provided in [Feature Flag Framework Runbook](https://cvsdigital.atlassian.net/wiki/spaces/VAE/pages/4111958109/Feature+Flag+Framework+Runbook#%F0%9F%8E%A2-Process---Access-Pattern).

· Choose between string vs boolean type for a feature flag:  
Each type has its own pro and con.  
String type allows more flexibility and extensibility to leverage the same feature flag for different configs (like Enum).  
Using boolean type limits the config to only two possible feature flag values - True Or False. It could be an appropriate option for designing a FF with single and straightforward use case.  
For example, if using string type for a feature flag of picking LLM provider, whenever there is a new provider or upgrade version available, the same feature flag could be used for rolling out and A/B testing. And only new revision required.

· When designing a rollout plan for A/B testing, progressively increase the rollout percentage; for example, 30% → 50% → 75% → 100%. Override list should remain the same if there is any, for consistent group assignment on A/B testing. Never decrease the rollout percentage.

## **2. Feature Flag Service**

· Set the proper TTL value for caching  
The idea here is to be configure caching TTL parameter to ensure compliance with the SLA.  
The current SLA for Feature Flag Framework is 10 mins:  
- FF Service is set with the default TTL as 5 minutes for caching FF values in-memory.  
- FF Client is set with the default TTL as 5 minutes for caching FF values in-memory.

## **3. Feature Flag Client Library**

· Maintain an Enum to list down all the feature flags used in the code. This will help tracking all the feature flags used in one place.

· # params.py

· from enum import Enum

· class FeatureFlags(Enum):

· LLM\_PROVIDER\_DATA\_FLAG = "llm\_provider" # JIRA link for flag removal

MULTI\_STEP\_API\_ROUTER\_FEATURE\_FLAG = "multi\_step\_api\_router" # JIRA link for flag removal

· Always create a single client object and reuse it across the application to effectively use the in-memory cache offered by the library. This should be a one-time creation per application.

· Do not read feature flags in application configurations, as the feature flags can depend on instance\_id (certain users or groups), which could be different for each event.

· default\_value: the value returned by the function only if the backend is unreachable. default\_value allows FF Framework to provide a degraded service instead of a full outage with the problem is worked out.

· When calling the get method for a feature flag in code,

o Always assign the returned feature flag value to a variable and use it for conditional check, which may avoid introducing new bug when removing the feature flag:

§ DO

§ from params import FeatureFlags

§

§ # Below approach 1 & 2 are RECOMMENDED when using feature flag

§ # Option 1 - assign flag value to a variable and use it for conditional check

§ flag\_name = FeatureFlags.LLM\_PROVIDER\_DATA\_FLAG.value

§ llm\_provider\_flag\_value = ff\_client.get\_flag\_string(flag\_name)

§ if llm\_provider\_flag\_value == "gemini":

§ answer("gemini")

§ else:

§ answer("openai")

§

§ # Option 2 - use the return value directly in-line with a match-case

§ flag\_name = FeatureFlags.LLM\_PROVIDER\_DATA\_FLAG.value

§ match ff\_client.get\_flag\_string(flag\_name):

§ case "gemini":

§ # new behavior

§ answer("gemini")

§ case \_:

§ # old behavior

answer("openai")

§ DON’T

§ # Below approach 3 & 4 are prone to bugs when removing feature flag.

§ # Hence NOT RECOMMENDED

§ # Option 1 - assign flag value to a variable and pass it as func parameter

§ flag\_name = FeatureFlags.LLM\_PROVIDER\_DATA\_FLAG.value

§ llm\_provider\_flag\_value = ff\_client.get\_flag\_string(flag\_name)

§ answer(llm\_provider\_flag\_value)

§

§ # Option 2 - use the return value directly in-line

§ flag\_name = FeatureFlags.LLM\_PROVIDER\_DATA\_FLAG.value

answer(ff\_client.get\_flag\_string(flag\_name))

o To allow minimum efforts when later removing this feature flag,

§ Call it as close as possible to code lines under development.

§ Avoid introducing any layers between where it being called and where the feature flag value is needed.

· Plan ahead and Remember to remove:  
A feature flag for a new feature will never reside permanently in the codebase. Once this new feature gets released and its developer knows it works after certain tests/period, its feature flag in the codebase should be removed.  
  
The principle is to always keep in mind that the cleanup PR should **ONLY** contain deleted lines. This is to ensure the cleanup PR doesn’t introduce any logic bugs. In practice, this may mean that the ‘previous’ and ‘new’ behaviors can have some code duplication between them, which is acceptable.  
  
Right after enabling a feature flag in the codebase, developers should use Jira to create a backlog for a follow-up task - to remove this feature flag. A link of this ticket should be in-line with where FF being called in the codebase. It should also be included in PR.  
  
**Tip:** It’s good for developers to setup a self-reminder for when to remove a feature flag from the codebase.

· # TODO: JIRA Link for feature flag removal

· flag\_value = ff\_client.get\_flag\_boolean(flag\_name)

· if (flag\_value):

· # new behavior

· else:

# old behavior

· When removing the feature flag  
The principle here is to always keep in mind that the cleanup PR should **ONLY** contain deleted lines.

· Note:

o Feature Flag Client parameter instance\_id:  
It identifies either a user, user group, or machine making the request. In most of the use cases, it identifies an individual user.  
It is important for providing a consistent user experience during progressive rollouts. Using this ID, Developer can force a FF value on a specific user group, regardless of the rollout percentage. Developer can also ensure during an A/B test, a single user does not have a feature toggle on/off sporadically.   
It could be potentially expanded beyond user level to a more abstract notion - endpoints like GPS vs demo UI etc. Further exploration is required.

· E2E Sample Usage

**Synchronous execution example**

############################################

# File: params.py

############################################

from iris\_feature\_flag\_client import FeatureFlagClient

from enum import Enum

# Create an Enum and keep track of all the feature flags used in the application

# along with the JIRA link next to it.

class FeatureFlags(Enum):

LLM\_PROVIDER\_DATA\_FLAG: "llm\_provider" # JIRA link for flag removal

MULTI\_STEP\_API\_ROUTER\_FEATURE\_FLAG = "multi\_step\_api\_router" # JIRA link for flag removal

# Create client object once and use it across the application

# In this example, creating the client object in params.py where we configure

# application based on environments

ff\_client = FeatureFlagClient(project\_name="csr-assistant")

###############################################################################

# File: app.py

###############################################################################

# Import client object from params.py to call get flag functions

from params import ff\_client, FeatureFlags

def simple\_function():

# TODO: JIRA link to follow-up feature flag removal

# Reading a feature flag of boolean type

flag\_name = FeatureFlags.MULTI\_STEP\_API\_ROUTER\_FEATURE\_FLAG.value

instance\_id = user\_id\_from\_context # user\_id from request which is stored in context variable

default\_value = False

# check if the multi\_step\_api\_router enabled for user\_id

flag\_value = ff\_client.get\_flag\_boolean(flag\_name, instance\_id, default\_value)

if flag\_value:

# new behavior

print(f"{flag\_name} is enabled.")

else:

# old behavior

print(f"{flag\_name} is disabled.")

# TODO: JIRA link to follow-up feature flag removal

# Reading a feature flag of string type

flag\_name = FeatureFlags.LLM\_PROVIDER\_DATA\_FLAG.value

instance\_id = request.user\_id # user\_id from the request

default\_value = "openai"

str\_flag\_value = ff\_client.get\_flag\_string(flag\_name, default\_value, default\_value)

# custom logic to process string flag value.

simple\_function()

**Asynchronous execution example**

###############################################################################

# File: params.py

###############################################################################

from iris\_feature\_flag\_client import AsyncFeatureFlagClient

from enum import Enum

# Create an Enum and keep track of all the feature flags used in the application

# along with the JIRA link next to it.

class FeatureFlags(Enum):

LLM\_PROVIDER\_DATA\_FLAG: "llm\_provider" # JIRA link for flag removal

MULTI\_STEP\_API\_ROUTER\_FEATURE\_FLAG = "multi\_step\_api\_router" # JIRA link for flag removal

# Create client object once and use it across the application

# In this example, creating the client object in params.py where we configure

# application based on environments

ff\_client\_async = AsyncFeatureFlagClient("csr-assistant")

###############################################################################

# File: app\_async.py

###############################################################################

# Import client object from params.py to call get flag functions

from params import ff\_client\_async, FeatureFlags

async def simple\_function():

# TODO: JIRA link to follow-up feature flag removal

# Reading a feature flag of boolean type

flag\_name = FeatureFlags.MULTI\_STEP\_API\_ROUTER\_FEATURE\_FLAG.value

instance\_id = user\_id\_from\_context # user\_id from request which is stored in context variable

default\_value = False

# check if the multi\_step\_api\_router enabled for user\_id

boolean\_flag\_value = await ff\_client\_async.get\_flag\_boolean(flag\_name, instance\_id, default\_value)

if boolean\_flag\_value:

print(f"{flag\_name} is enabled.")

else:

print(f"{flag\_name} is disabled.")

# TODO: JIRA link to follow-up feature flag removal

# Reading a feature flag of string type

flag\_name = FeatureFlags.LLM\_PROVIDER\_DATA\_FLAG.value

instance\_id = request.user\_id # user\_id from the request

default\_value = "openai"

str\_flag\_value = await ff\_client\_async.get\_flag\_string(flag\_name, default\_value, default\_value)

# custom logic to process string flag value.

await simple\_function()

# **Limitation**

· Size of a Feature Flag:  
Keep it under 100 MB. It cannot hold configuration values that are more than 256 MB.  
The project and FF name must be 4 KB or less. They are part of [Bigtable Best Practice](https://cloud.google.com/bigtable/docs/schema-design#best-practices).