AWS Machine Learning Blog

Dynamic A/B testing for machine learning models with Amazon SageMaker MLOps projects

by Julian Bright | on 09 JUL 2021 | in Amazon Machine Learning, Amazon SageMaker, Amazon SageMaker JumpStart, Artificial Intelligence | Permalink | Comments | Share

In this post, you learn how to create a MLOps project to automate the deployment of an Amazon SageMaker endpoint with multiple production variants for A/B testing. You also deploy a general purpose API and testing infrastructure that includes a multi-armed bandit experiment framework. This testing infrastructure will automatically optimize traffic to the best-performing model over time based on user feedback.

Amazon SageMaker MLOps projects are a new capability recently released with Amazon SageMaker Pipelines, the first purpose-built, easy-to-use, continuous integration and continuous delivery (CI/CD) service for ML. The MLOps project template provisions the initial setup required for a complete end-to-end MLOps system, including model building, training, and deployment, and can be customized to support your own organizations requirements.

When deploying new models to production, it's often a good idea to gradually roll out new models to users as part of an A/B testing experiment in which you monitor a high-level metric, such as click-through rate or conversion rate, to measure if your new model is an improvement over your previously deployed production variant.

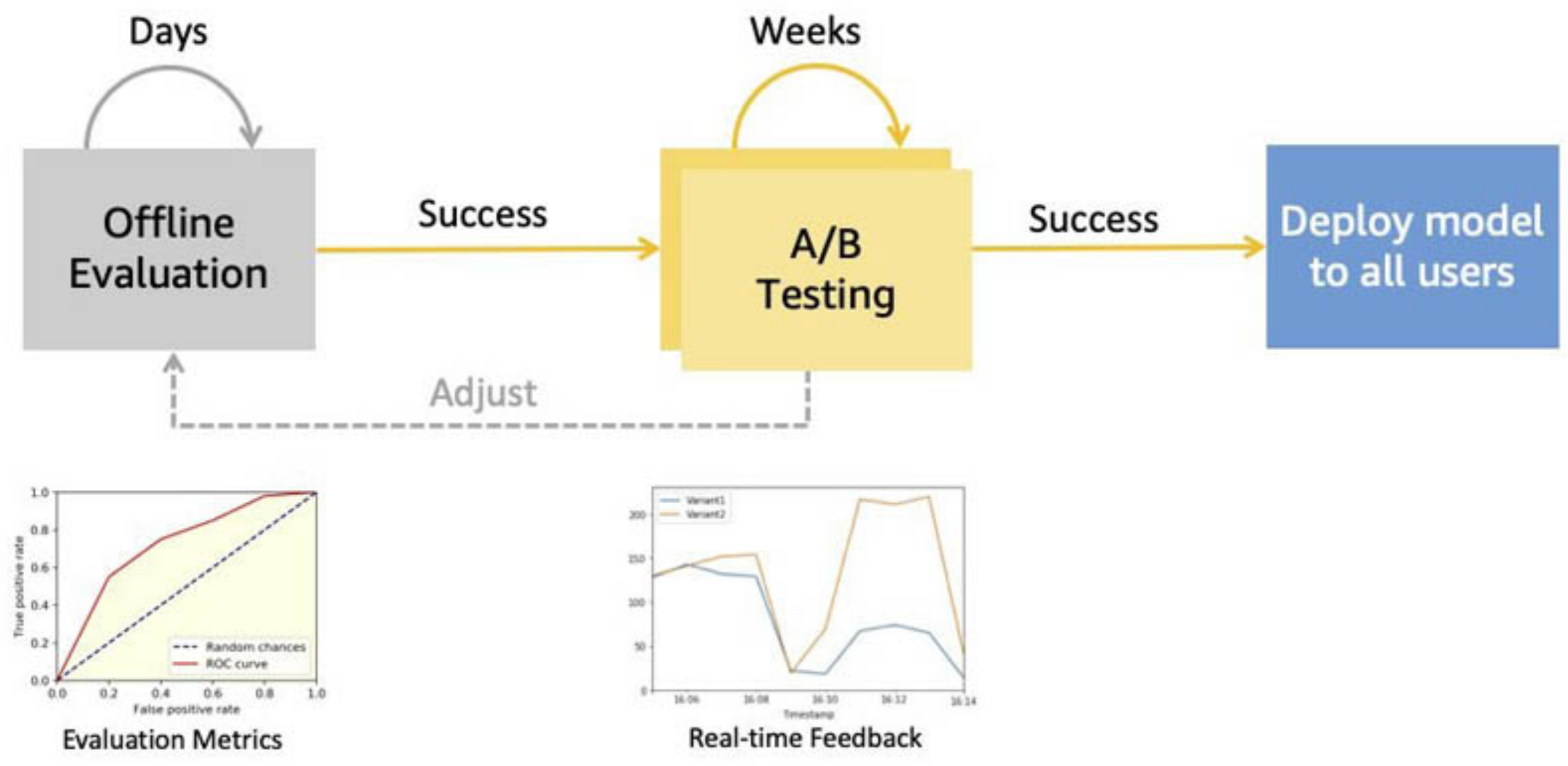
This post contains three sections:

- An introduction to A/B testing design to provide a high-level overview of the concepts and algorithms
- Hands-on steps for operationalizing an A/B testing deployment pipeline
- A simulation of an A/B test against your deployed machine learning (ML) model variants

A/B testing for machine learning

In the context of ML, performing A/B testing on the new model and the old model with production traffic can be an effective final step after offline evaluation because in real life many things influence user behavior and metrics. By randomizing which users are in which group (A or B), you minimize the chances that other factors, like mobile vs. desktop, drive your metrics.

Offline evaluation is a data science practice of holding out a test dataset during ML training to evaluate how effective your model is at performing predictions on unseen data. Data scientists iterate on the evaluation process to improve performance, tuning the data preparation and model parameters, using visualizations such an ROC curve to measure a binary classifier's ability to correctly separate true from false positive predictions.



Before deploying this model to all users, it's a good idea to run this new or "challenger" model side-by-side with an existing "champion" model in an A/B test to find empirical evidence of the impact this new model has on your business metrics, such as click-through rate, conversion rate, or revenue. By collecting real-time feedback as your model is running, you can optimize how traffic is distributed between the champion and challenger models of the period of the test, which can often run for several weeks.

When you're confident that this new challenger model is the outperforming your previous model, you can deploy this new model to all users, and begin the process again.

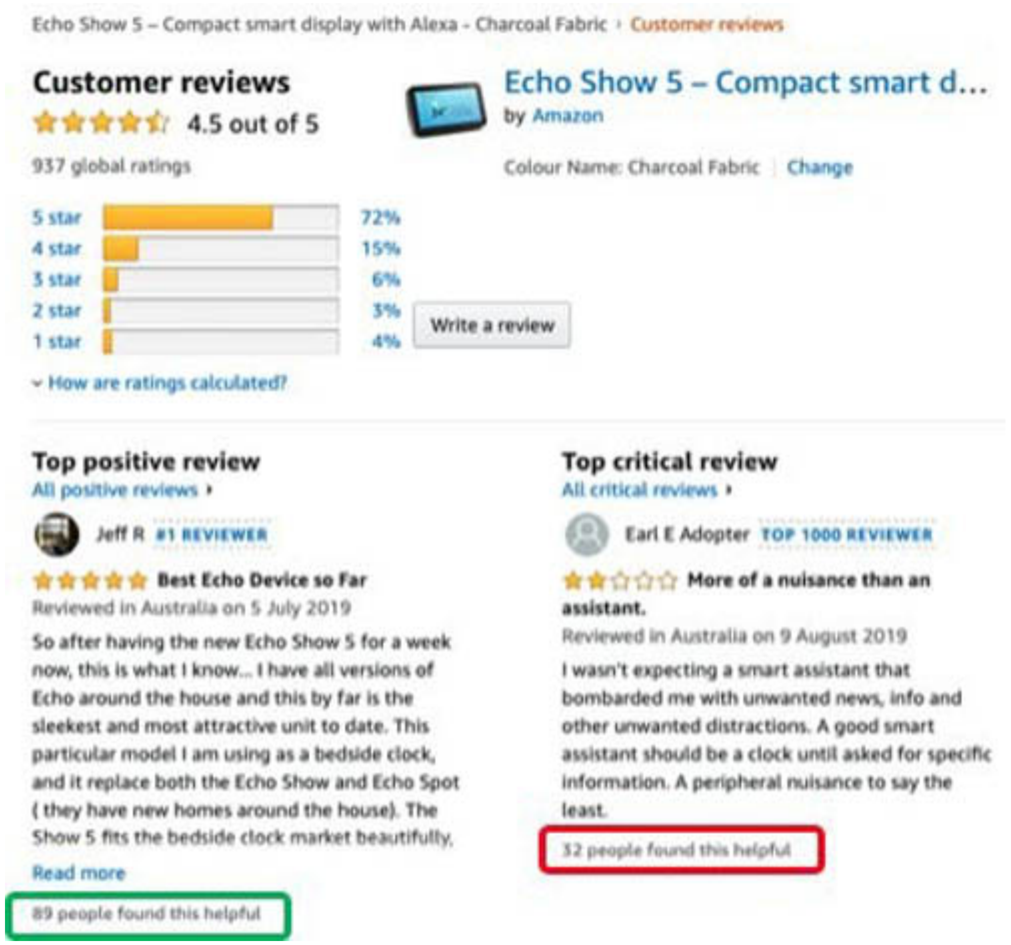
When should you test?

After you identify a clear business metric that you can measure based on regular user feedback, you should consider A/B testing when a change to your dataset or model definition occurs. The following table summarizes some examples.

Different Dataset	**Different Model**
Dataset has been updated to include latest fresh data	You're trying a different algorithm architecture
Dataset has been cleaned, normalized, or scaled differently	You're experimenting with different hyperparameters
Datasets has been resampled to remove bias or adjust for minority class imbalance	You're using transfer learning to fine-tune a pre-trained model

Use case: Recommending helpful reviews

In this post, you build a review helpfulness binary classifier trained on the Amazon Customer Reviews Dataset using the SageMaker Blazing Text algorithm. The following screenshot shows the product page for the Amazon Echo Show 5, which has 937 reviews with an average star rating of 4.5 out of 5. The page also shows a top positive and top critical review.


Echo Show 5 – Compact smart display with Alexa - Charcoal Fabric › Customer reviews

Customer reviews

4.5 out of 5

937 global ratings

Echo Show 5 – Compact smart d...

by Amazon

Colour Name: Charcoal Fabric | Change

5 star	72%
4 star	15%
3 star	6%
2 star	3%
1 star	4%

Write a review

⌄ How are ratings calculated?

Top positive review

All positive reviews ›

Jeff R #1 REVIEWER

Best Echo Device so Far

Reviewed in Australia on 5 July 2019

So after having the new Echo Show 5 for a week now, this is what I know... I have all versions of Echo around the house and this by far is the sleekest and most attractive unit to date. This particular model I am using as a bedside clock, and it replace both the Echo Show and Echo Spot (they have new homes around the house). The Show 5 fits the bedside clock market beautifully.

Read more

89 people found this helpful

Top critical review

All critical reviews ›

Earl E Adopter TOP 1000 REVIEWER

More of a nuisance than an assistant.

Reviewed in Australia on 9 August 2019

I wasn't expecting a smart assistant that bombarded me with unwanted news, info and other unwanted distractions. A good smart assistant should be a clock until asked for specific information. A peripheral nuisance to say the least.

32 people found this helpful

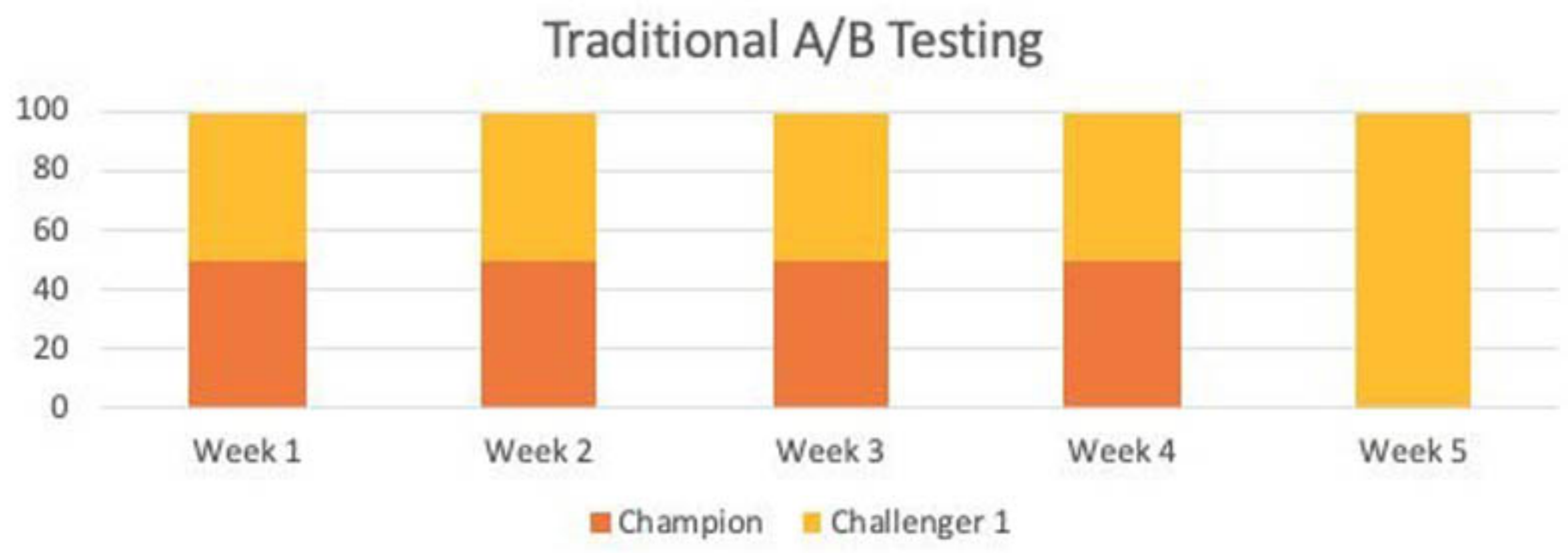
Users looking to buy an Echo Show (or any product on Amazon) provide feedback on reviews that are helpful, and we use this data to train an ML classification model that can then identify the most helpful reviews based on just the free text. This allows us to surface the best new reviews from the comments while we wait for users to validate our selection with their feedback.

Apply A/B testing

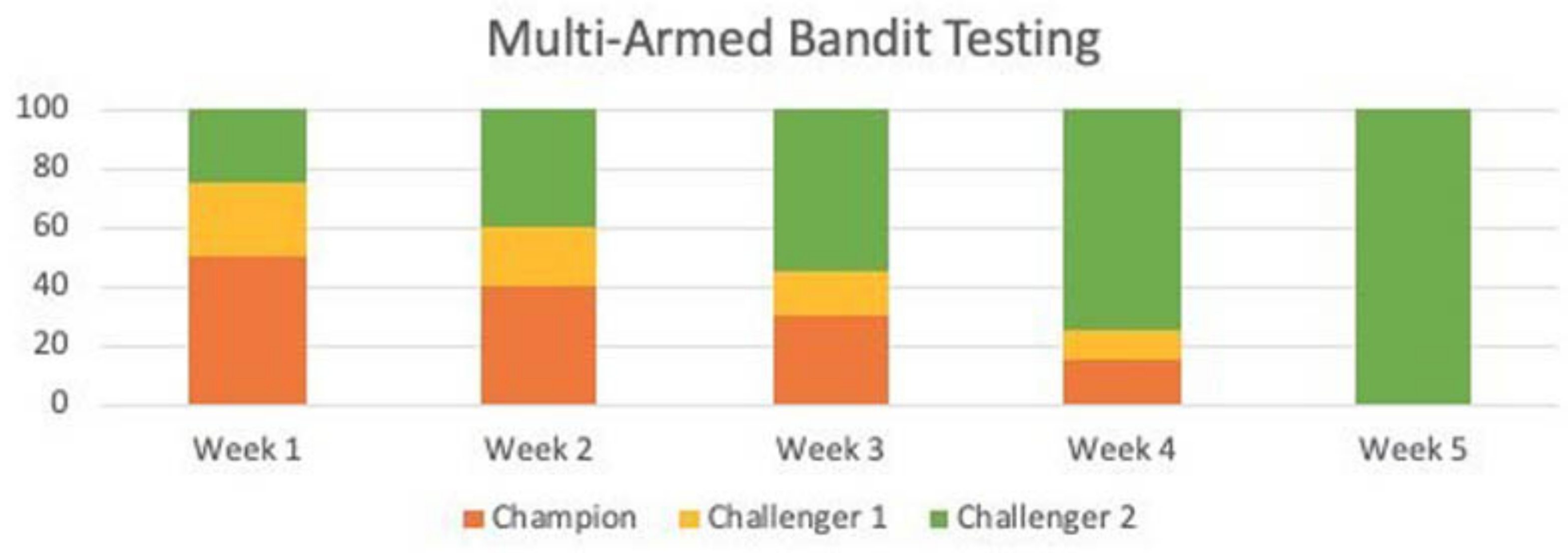
In this section, we deep dive into A/B testing. For more details on how these algorithms are implemented in Python, check out the source code in the GitHub repo.

For an A/B test to be considered successful, you need to perform a statistical analysis of the metrics gathered from the test to determine if there is a statistically significant result. This analysis is based on the significance level you set for the experiment; a 5% significance level is considered the industry standard. For example, a significance level of 0.05 indicates a 5% risk of concluding that a difference exists when there is no actual difference. A lower significance level means that we need stronger evidence for a statistically significant result. For more information about statistical significance, see A Refresher on Statistical Significance.

Traditional A/B testing runs for a defined period based on the number of users necessary to reach a statistically significant result. Tools such as Evan Miller's Awesome A/B Tools can help you determine how large your sample size needs to be. During this initial period of exploration, you evaluate whether your new model variant is going to challenge the current champion, sending traffic to the less effective variant until the test is complete, which in this case is not until week 5 (as shown in the following graph).



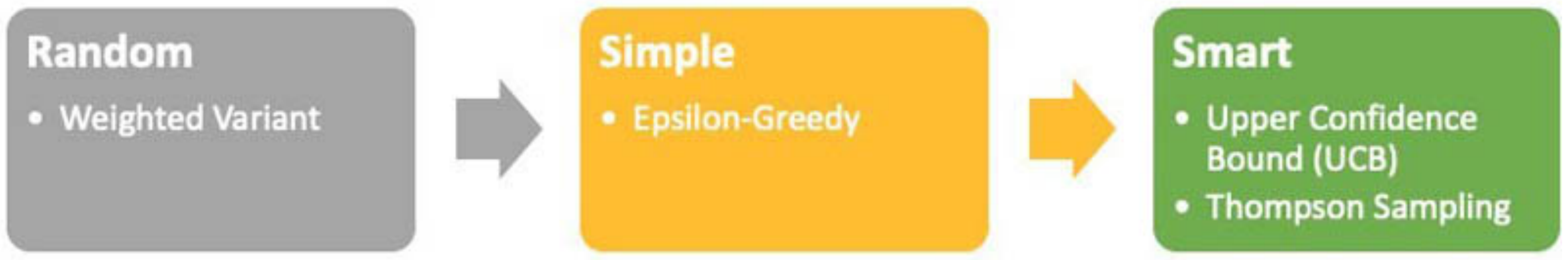
Multi-armed bandit testing is dynamic, and includes a gradual change from exploration to exploitation over the duration of the test, sending more traffic to the challenger variant that is delivering the highest reward as defined by your conversion metric (as shown in the following graph). This reduces the traffic being sent to the less effective variant over the lifetime of the test.



A/B testing strategies

For an A/B Test to be effective, users need to be assigned to a particular model variant for the duration of the experiment.

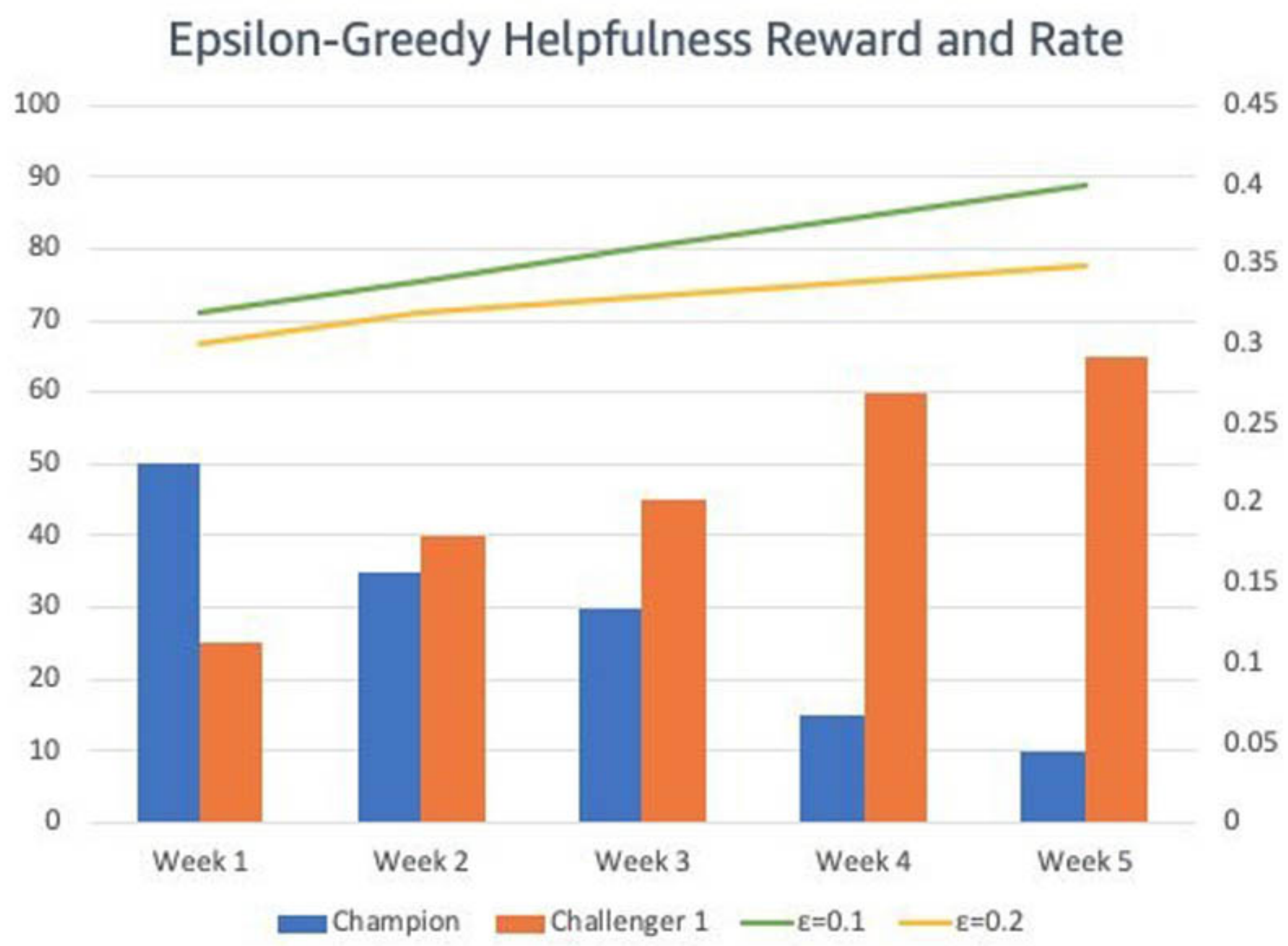
We start assignment using a random distribution based the initial model weights while we collect user feedback as invocation and conversion metrics. We can use reward probability estimates based on the user feedback we collect to exploit the best-performing model variants with either the simple Epsilon-Greedy bandit strategy, or a more sophisticated strategy such as upper confidence bound or Thompson sampling.



Epsilon-Greedy

The simple ε-greedy algorithm selects the best variant most of the time, but does random exploration occasionally:

- If the ε parameter is `0.1` then 10% of the time, we choose a model variant at random. The other 90% of the time, we choose the variant that has the highest expectation of reward. In the following example, this leads us to choosing the better-performing Challenger 1 model more often.
- When the ε parameter is `0.2`, we explore 20% of the time, introducing more chance of selecting the poorer-performing variant, which is why the reward rate is lower overall, as shown with the yellow line.



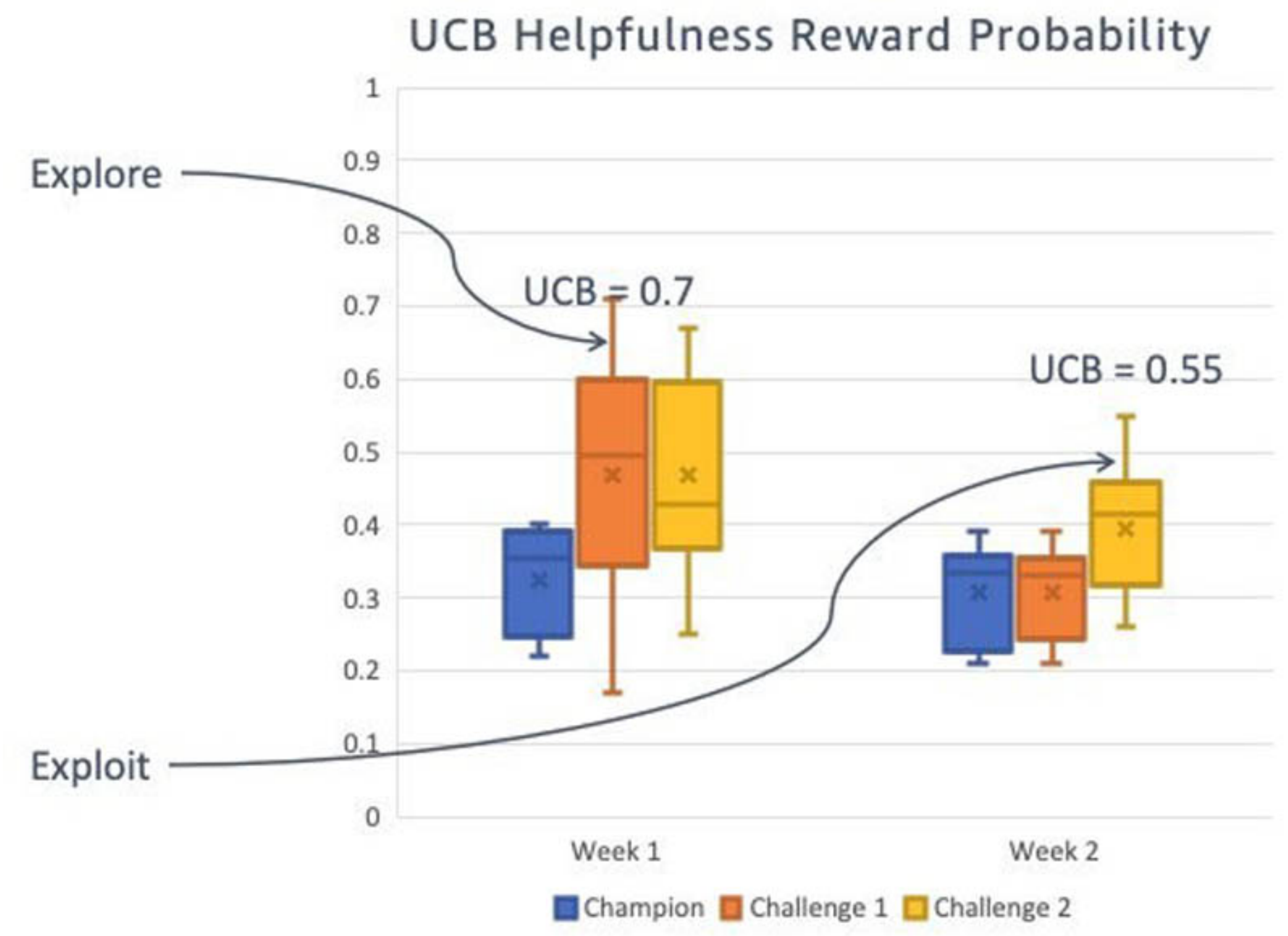
Upper confidence bound

The upper confidence bound (UCB) algorithm introduces uncertainty around variants by keeping track of how many times a variant is explored:

- For each variant invocation `i`, we record the average reward $\bar{\mu}_i$ and number of times we tried it n_i. `t` is the total number of invocations for all variants. The UCB1 algorithm formula is

$$\bar{\mu}_i + \sqrt{\frac{2log(t)}{n_i}}$$

- We explore variants with high uncertainty that are infrequently selected. In the following example, the Challenger 1 variant is more likely to be selected at the end of week 1 due to a higher upper confidence bound of `0.7`.
- In week 2 as uncertainty levels drop, we exploit the variants with the highest mean plus uncertainty, which puts the Challenger 2 variant ahead with an upper confidence bound of `0.55`.

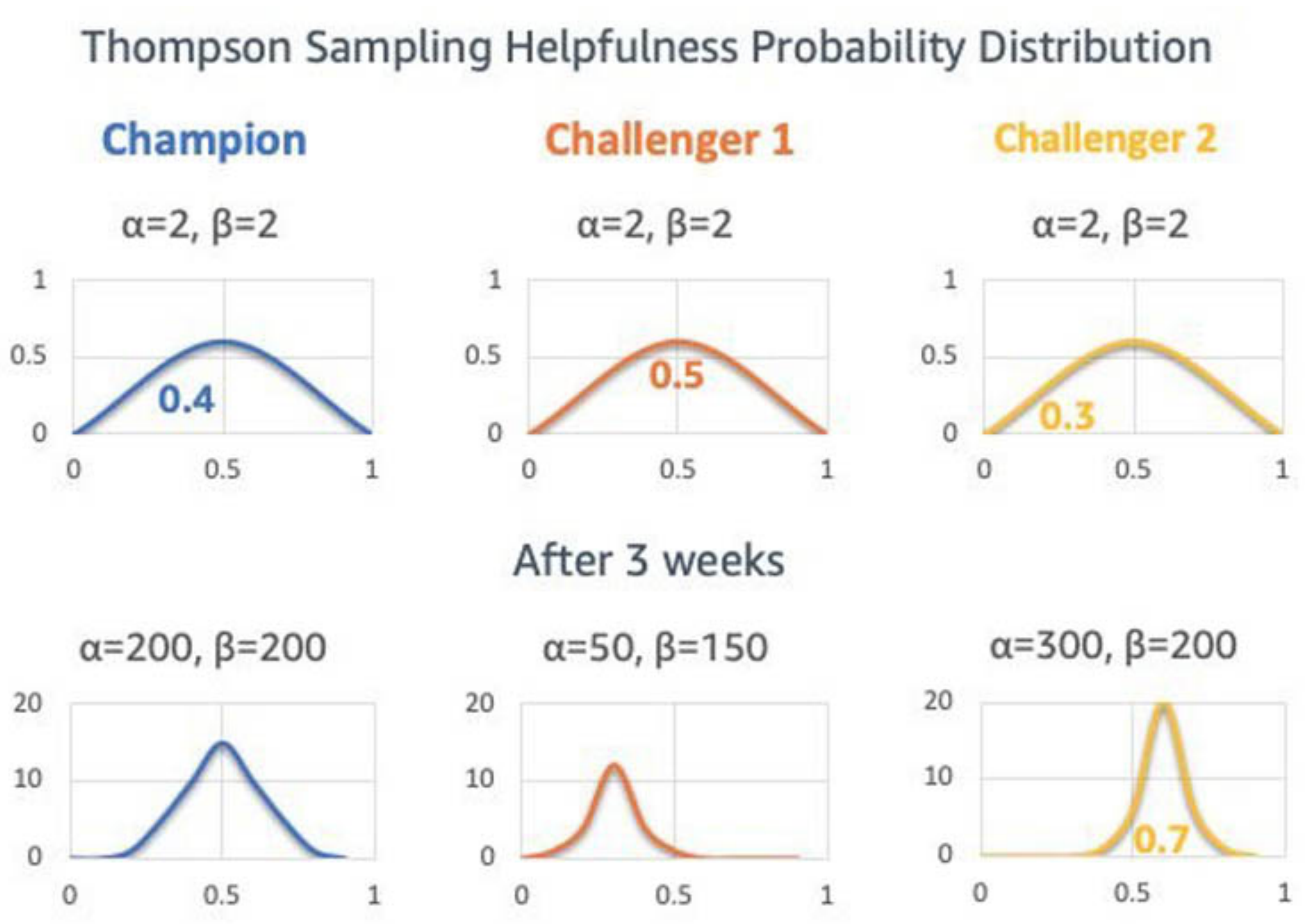


Thompson sampling

Thompson sampling estimates the uncertainty around variants by using beta probability distributions:

- Beta probability distributions are defined by parameters α and β, which correspond to helpful and not helpful reviews, respectively.

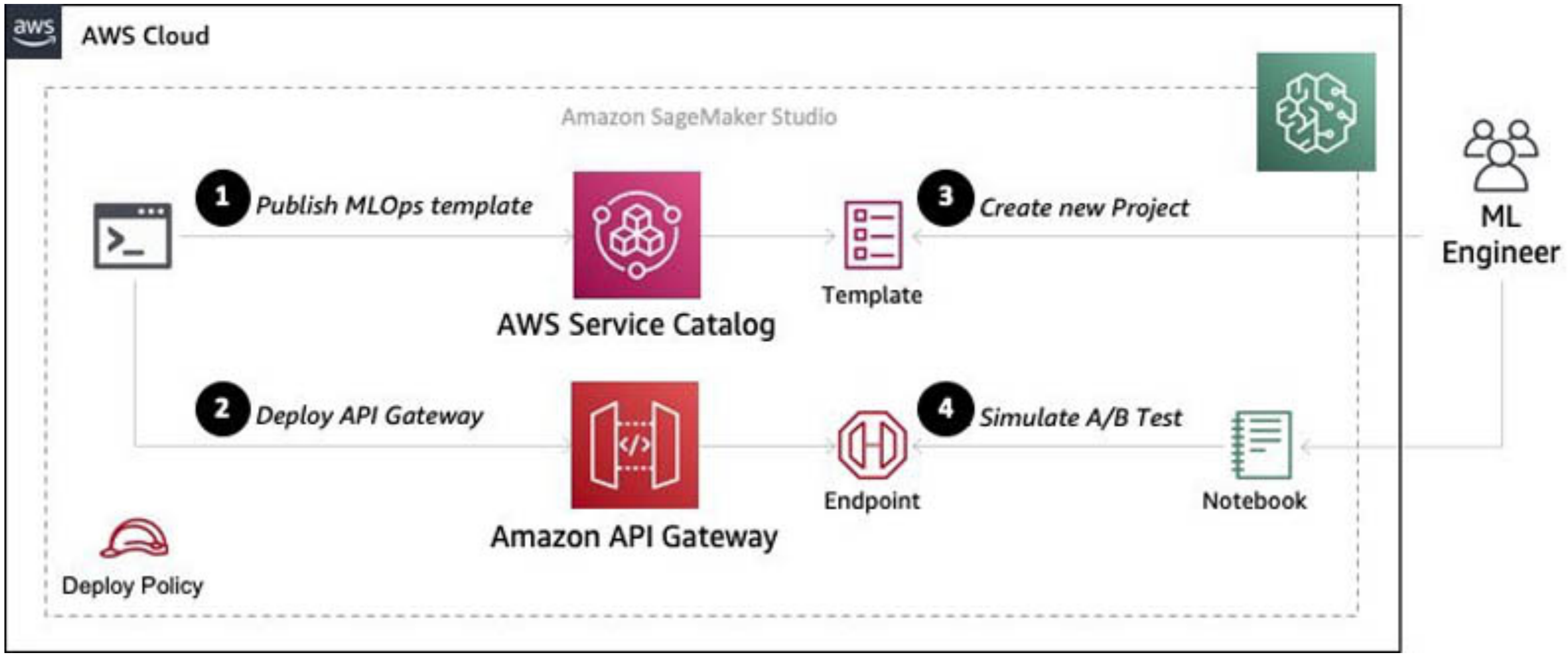
- We explore variants by randomly sampling a distribution for each variant, from which we select the variant with the highest sampled value. In the following initial example, we select Challenger 1 with a value of 5.
- As we capture user feedback, we update the `α` and `β` values, which adjusts the shape of the distributions. As you can see, the skew right in Challenger 2 for week 3 has increased the number of helpful reviews.



In the next section, we explore the solution for implementing A/B testing with SageMaker.

Solution overview

The following diagram illustrates the architecture for our solution.



To build this solution in Amazon SageMaker Studio, we use the AWS Cloud Development Kit (AWS CDK). If you're new to AWS CDK, we recommend that you start your journey with the AWS CDK Workshop for Python. For more information, see AWS CDK Reference Documentation.

The following are the high-level steps to deploy this solution:

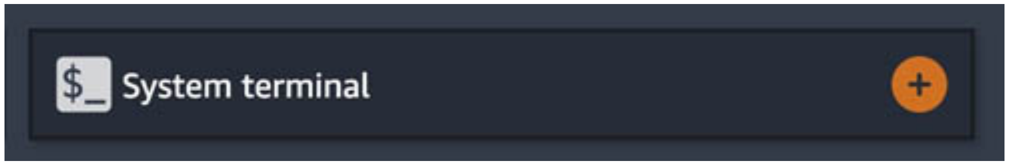
1. Publish a SageMaker MLOps Project template in the AWS Service Catalog.
2. Deploy an Amazon API Gateway and testing infrastructure.
3. Create a new project in Studio.

Then you can train and deploy ML models for A/B testing in the sample notebook provided.

Getting started

Amazon SageMaker MLOps project templates are defined as AWS CloudFormation and published via the AWS Service Catalog. These are made available to data scientists via Studio, an IDE for ML. To configure Studio in your account, complete the following steps:

1. Prepare your Studio domain.
2. Enable SageMaker project templates and SageMaker JumpStart for this account and Studio users.
3. Open Studio.
4. In the Launcher, under **Utilities and files**, choose **System terminal**.



5. Clone the GitHub repository in this new terminal:

```
git clone https://github.com/aws-samples/amazon-sagemaker-ab-testing-pipeline.git
cd amazon-sagemaker-ab-testing-pipeline
```

For instructions on installation prerequisites and how to configure permissions required for AWS CDK, see the GitHub repo README file.

6. When your environment is set up, and you can list the AWS CDK stacks using the following command:

```
cdk list
```

You're ready to move on to the next steps.

Publish the SageMaker MLOps project template

In this step, you create a portfolio and product to provision a custom SageMaker MLOps project template in the AWS Service Catalog and configure it so you can launch the project from within your Studio domain.

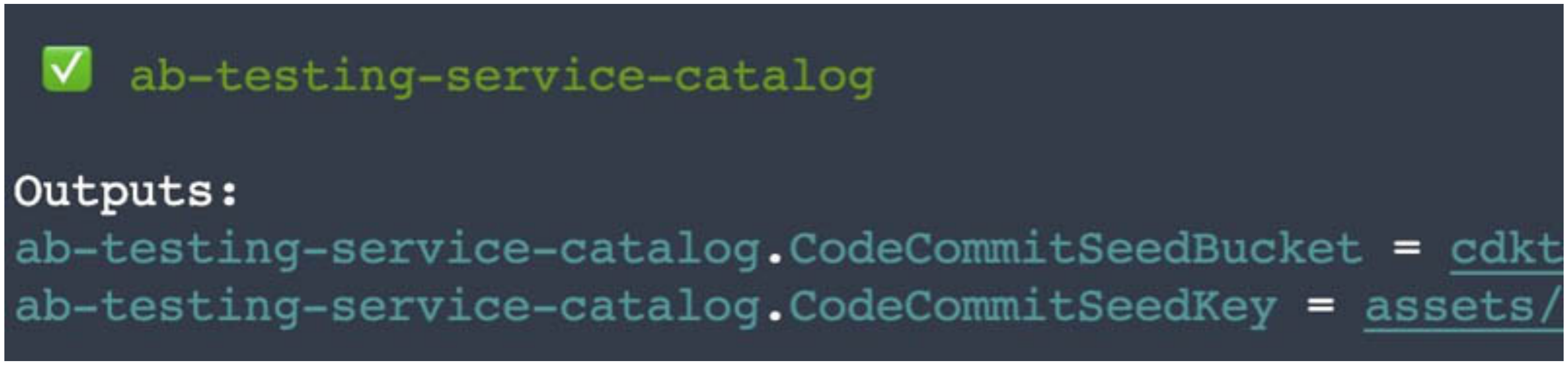
Run the following command to deploy the MLOps project template, passing the required `ExecutionRoleArn` parameter:

```
cdk deploy ab-testing-service-catalog \
    --parameters ExecutionRoleArn=<<sagemaker-studio-execution-role>
```

If you don't have an AWS Identity and Access Management (IAM) execution role available, go to the SageMaker console and choose **Amazon SageMaker Studio**. In the Studio **Summary** section, locate the attribute **Execution role**. Search for the name of this role in IAM to copy the ARN.

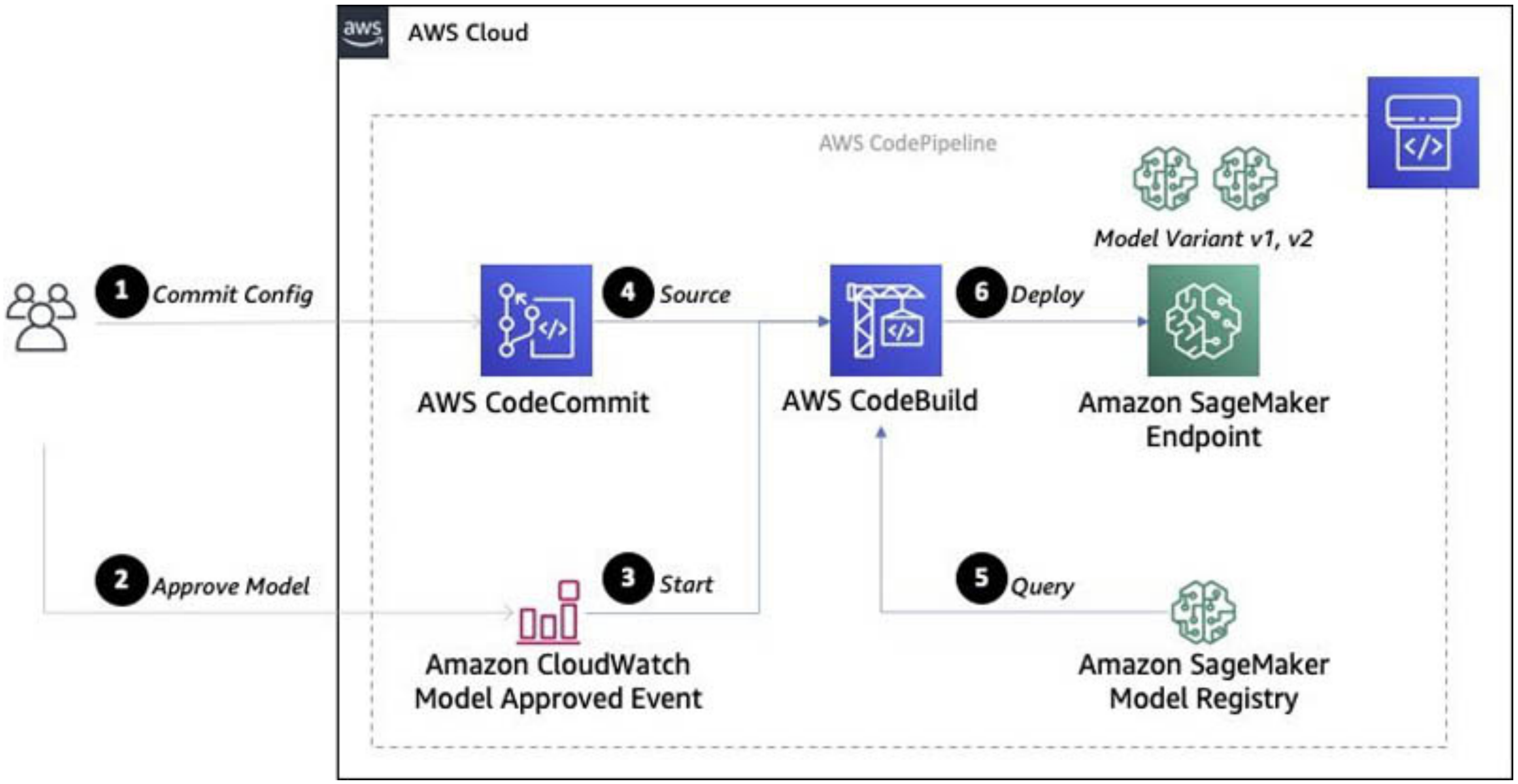
AWS CDK lists the changes and asks you to confirm you wish to deploy these. Enter y for yes.

This stack uploads the deployment pipeline code to Amazon Simple Storage Service (Amazon S3) and returns the outputs `CodeCommitSeedBucket` and `CodeCommitSeedKey`, which you need when creating the Studio project.



```
✅  ab-testing-service-catalog

Outputs:
ab-testing-service-catalog.CodeCommitSeedBucket = cdkt
ab-testing-service-catalog.CodeCommitSeedKey = assets/
```

The MLOps project template creates a deployment pipeline that is triggered when a new model is approved, as shown in the following diagram.



The following are the sequence of events that occur when a model is approved in the SageMaker model registry:

1. The data scientist commits a configuration file to **AWS CodeCommit** that includes the stage and A/B testing strategy.
2. The data scientist approves the model in the SageMaker model registry.
3. An **Amazon CloudWatch** model approved event starts the **AWS CodeBuild** job (app.py).
4. CodeBuild pulls the latest source from CodeCommit.
5. CodeBuild queries the SageMaker model registry for the latest champion and challenger models (model_registry.py)
6. CodeBuild outputs the CloudFormation stack (sagemaker_stack.py) to deploy the SageMaker endpoint.

After an endpoint is in service, it's available to be served by the API, as you see in the next section.

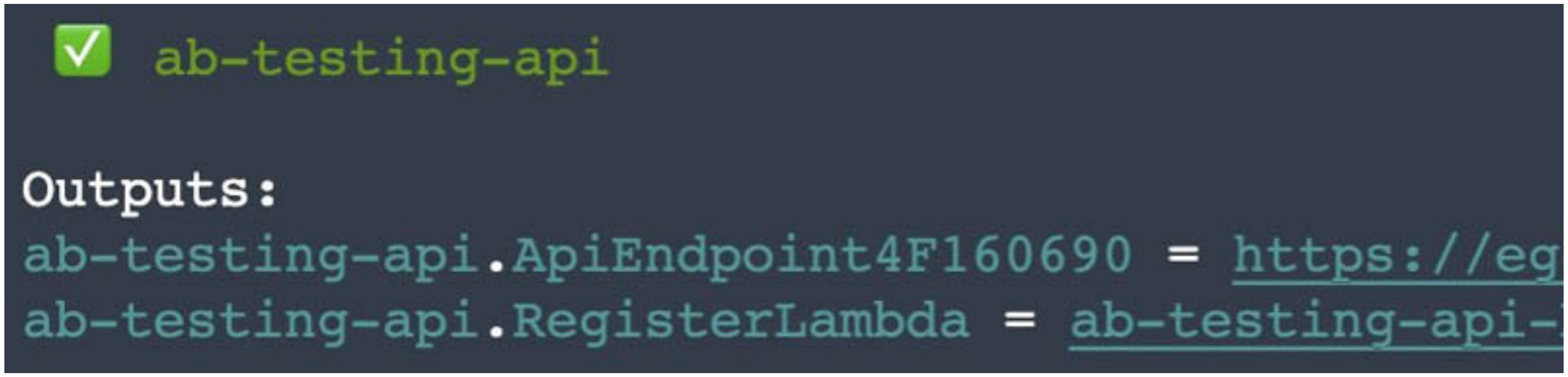
Deploy the API and testing infrastructure

In this step, you deploy an API Gateway and supporting resources to enable dynamic A/B testing of any SageMaker endpoint that has multiple production variants.

Run the following command to deploy the API and testing infrastructure with optional configuration:

```
cdk deploy ab-testing-api -c stage_name=dev
```

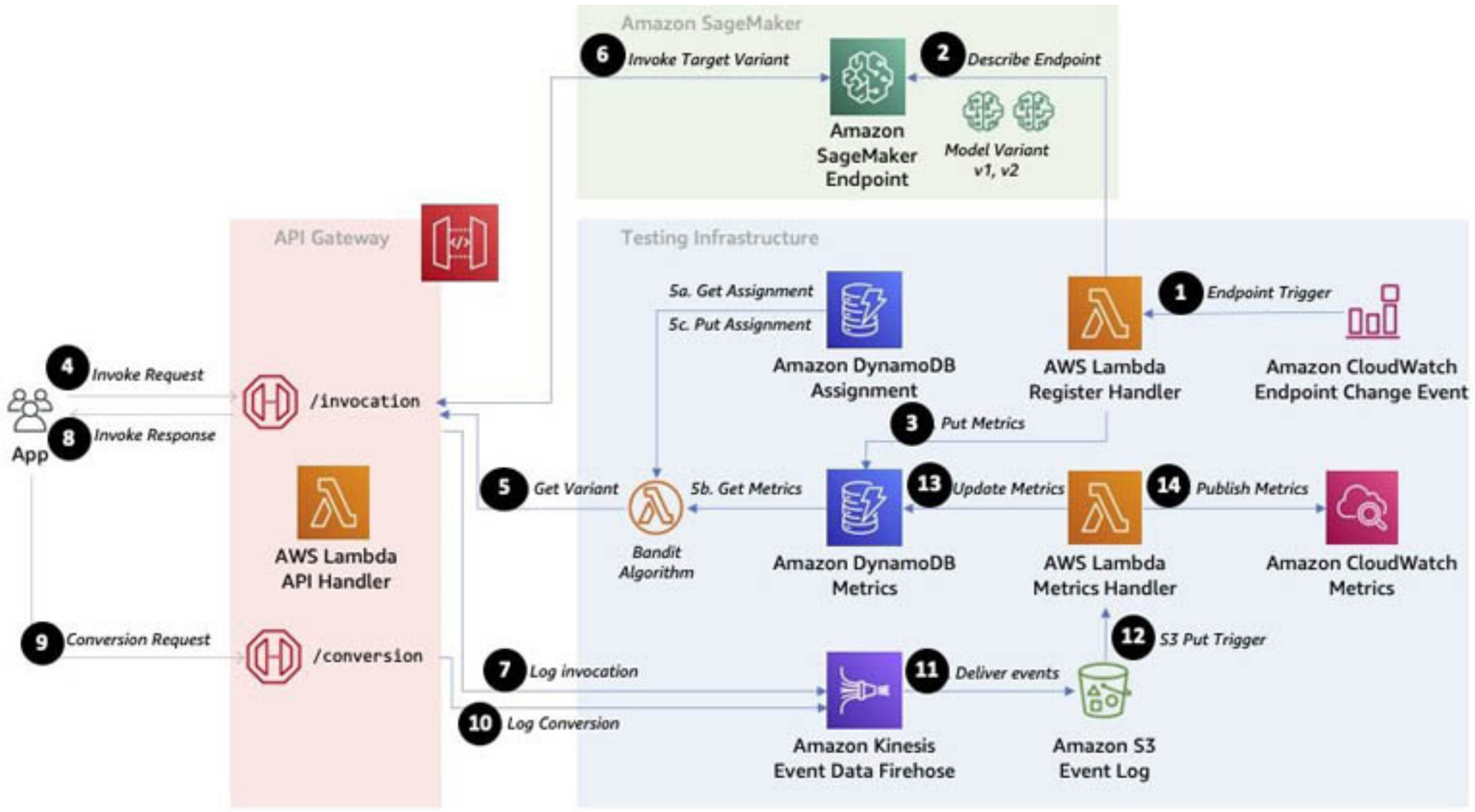
This stack outputs an `ApiEndpoint` URL, which you provide to the A/B Testing sample notebook.



```
✅  ab-testing-api

Outputs:
ab-testing-api.ApiEndpoint4F160690 = https://eg
ab-testing-api.RegisterLambda = ab-testing-api-
```

The API provides a wrapper around SageMaker to dynamically invoke the model variant assigned to a user.

The following workflow starts when a SageMaker endpoint status changes to be `InService`.



This workflow includes the following steps:

1. A CloudWatch endpoint change event triggers the **AWS Lambda** register handler (lambda_register.py).
2. The register handler queries the SageMaker endpoint to get model variant weights and configuration.
3. The register handler updates an **Amazon DynamoDB** metrics table to set variant weights and clear any metrics for the endpoint.

The app is now available to make an inference invocation against this endpoint.

4. The app sends an invocation request to the API specifying the user, endpoint name, and inference payload.
5. The API handler (lambda_invoke.py) attempts to get the assigned variant from the DynamoDB assignment table for the given user and endpoint. If

no variant is found, metrics for the endpoint are retrieved from the DynamoDB metrics table and passed to the bandit algorithm (algorithm.py). The bandit algorithm returns a new variant for the user, which is written back to the DynamoDB assignment table.

6. The API handler invokes the target variant on the SageMaker endpoint with the inference payload.
7. The API handler logs the invocation to **Amazon Kinesis Data Firehose**.
8. The API handler returns the invocation response to the app, including the assigned user variant.

After a successful conversion action from the user, the app calls the API to update metrics.

9. The app sends a conversion request to the API specifying the user and endpoint name.
10. The API handler logs the conversion event for the user and variant to Kinesis Data Firehose.

Periodically, events are delivered to Amazon S3 as a series of JSON lines.

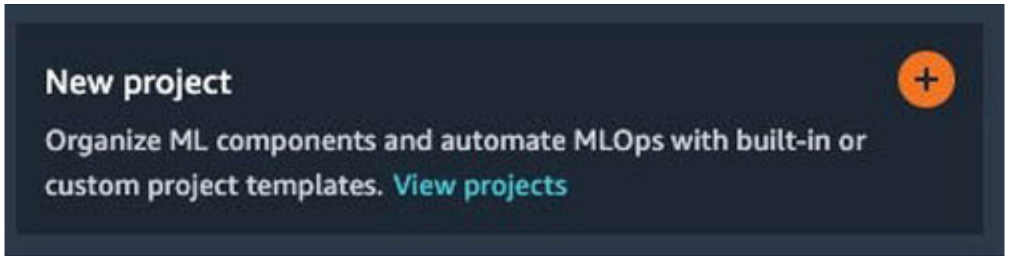
11. The Firehose delivery stream delivers its events to Amazon S3.
12. The put to Amazon S3 triggers the Lambda metrics handler passing the source object.
13. The metrics handler (lambda_metrics.py) reads the S3 object, groups events by endpoint, and updates metrics in the DynamoDB metrics table.
14. The metrics handler publishes metrics to CloudWatch.

With the MLOps template published and our API infrastructure ready, we can continue.

Create a new project in Studio

When your MLOps project template is registered in the AWS Service Catalog, you can create a project using your new template.

1. Switch back to the Launcher in Studio.
2. Choose **New project** in the **ML tasks and components**

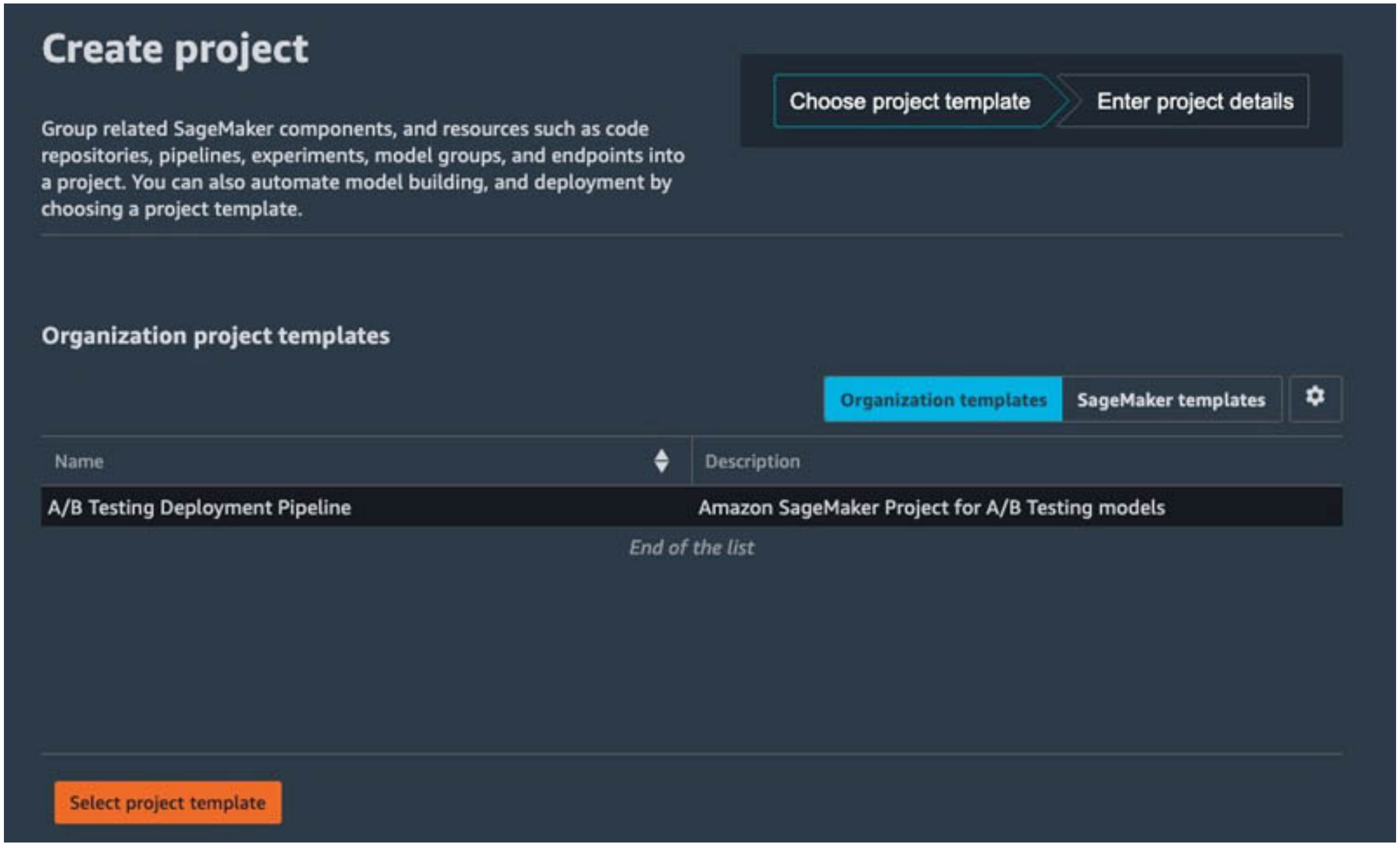


Organize ML components and automate MLOps with built-in or custom project templates.

On the **Create project** page, **SageMaker templates** is chosen by default. This option lists the built-in templates. However, you should use the template you published for the A/B testing deployment pipeline.

3. Choose **Organization templates**.
4. Choose **A/B Testing Deployment Pipeline**.
5. Choose **Select project template**.

You may need to refresh the Studio IDE to see the latest organization templates.



Create project

Group related SageMaker components, and resources such as code repositories, pipelines, experiments, model groups, and endpoints into a project. You can also automate model building, and deployment by choosing a project template.

Organization project templates

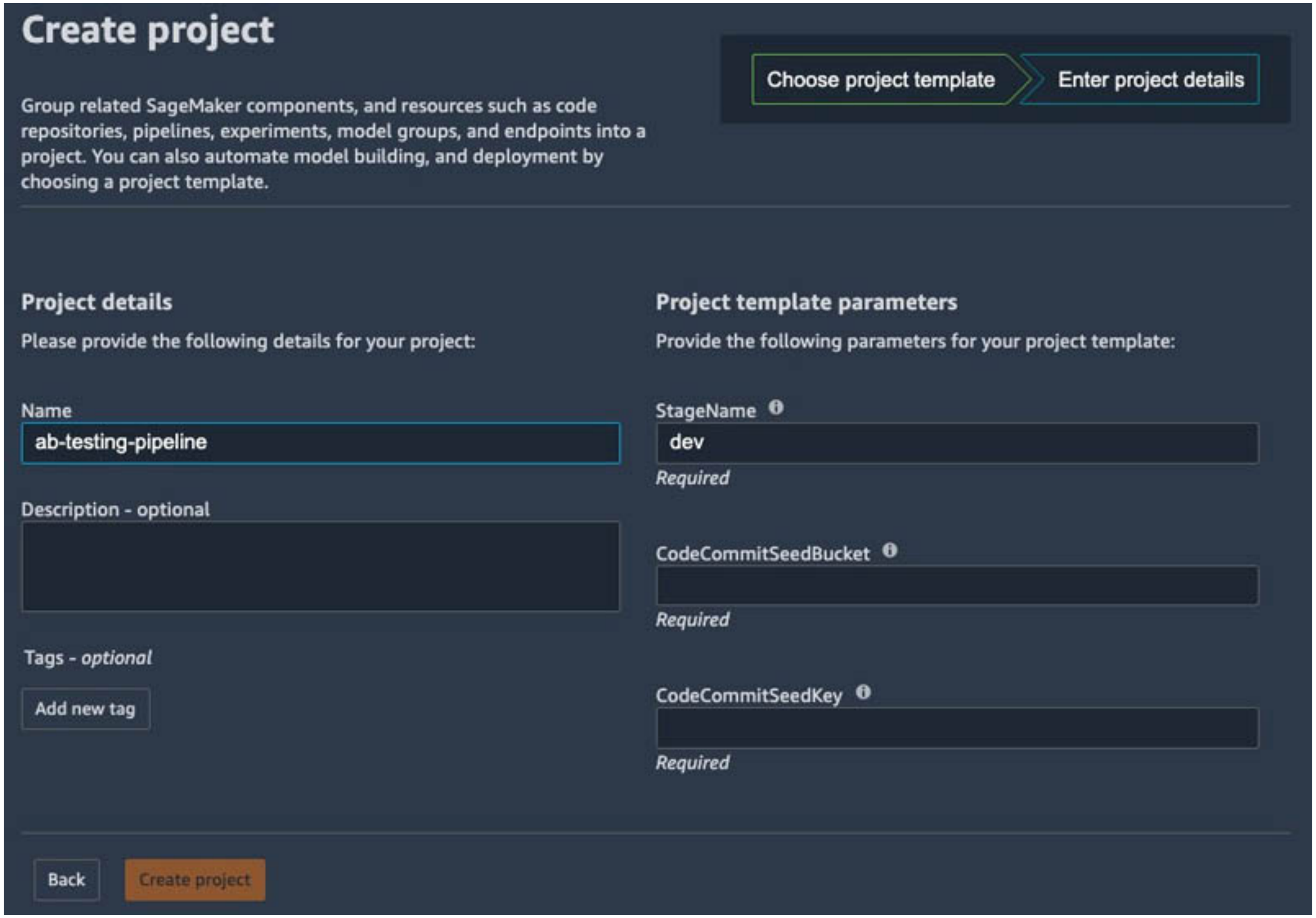
Name	Description
A/B Testing Deployment Pipeline	Amazon SageMaker Project for A/B Testing models

End of the list

6. In the **Project details** section, for **Name**, enter `ab-testing-pipeline` .

The project name must have 32 characters or fewer.

7. In the Project template parameters, for **StageName**, leave the default `dev` .
8. For **CodeCommitSeedBucket**, enter the `CodeCommitSeedBucket` output from the `ab-testing-service-catalog` stack.
9. For **CodeCommitSeedKey**, enter the `CodeCommitSeedKey` output from the `ab-testing-service-catalog` stack.
10. Choose **Create project**.


Create project

Group related SageMaker components, and resources such as code repositories, pipelines, experiments, model groups, and endpoints into a project. You can also automate model building, and deployment by choosing a project template.

Project details

Please provide the following details for your project:

Name: ab-testing-pipeline

Description - optional

Tags - optional

Project template parameters

Provide the following parameters for your project template:

StageName: dev
Required

CodeCommitSeedBucket
Required

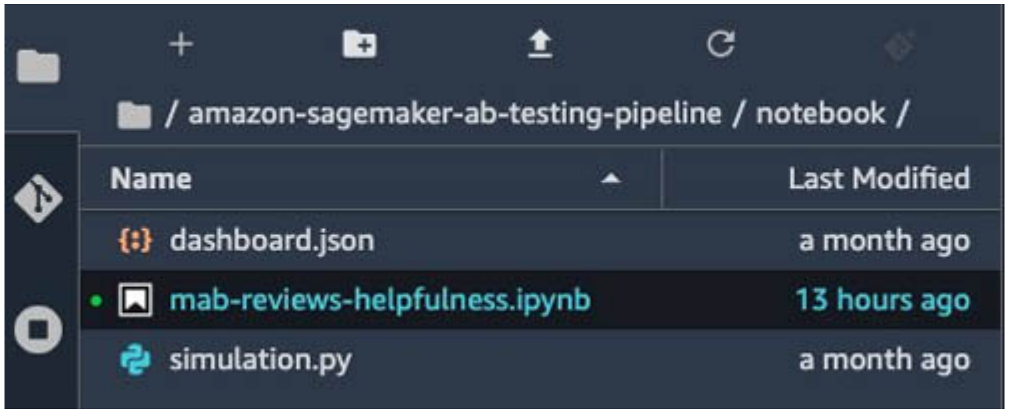
CodeCommitSeedKey
Required

This takes a few minutes to provision the project, in the meantime we can move on to training our models.

Train and deploy ML models for A/B testing

In the following section, you learn how to train, deploy, and simulate a test against our A/B testing pipeline using the mab-review-helpfulness.ipynb sample notebook from the GitHub repository.

Start by browsing to the `amazon-sagemaker-ab-testing-pipeline` folder you cloned from GitHub, navigate to the `notebook` directory, and open the Studio notebook named `mab-reviews-helpfulness.ipynb`.


/ amazon-sagemaker-ab-testing-pipeline / notebook /

Name	Last Modified
dashboard.json	a month ago
mab-reviews-helpfulness.ipynb	13 hours ago
simulation.py	a month ago

This notebook takes you through several steps:

1. Prepare your data.
2. Run a SageMaker pipeline.
3. Run your tuning job.
4. Test the endpoint.
5. Run your A/B testing simulation.
6. Determine the winning model.

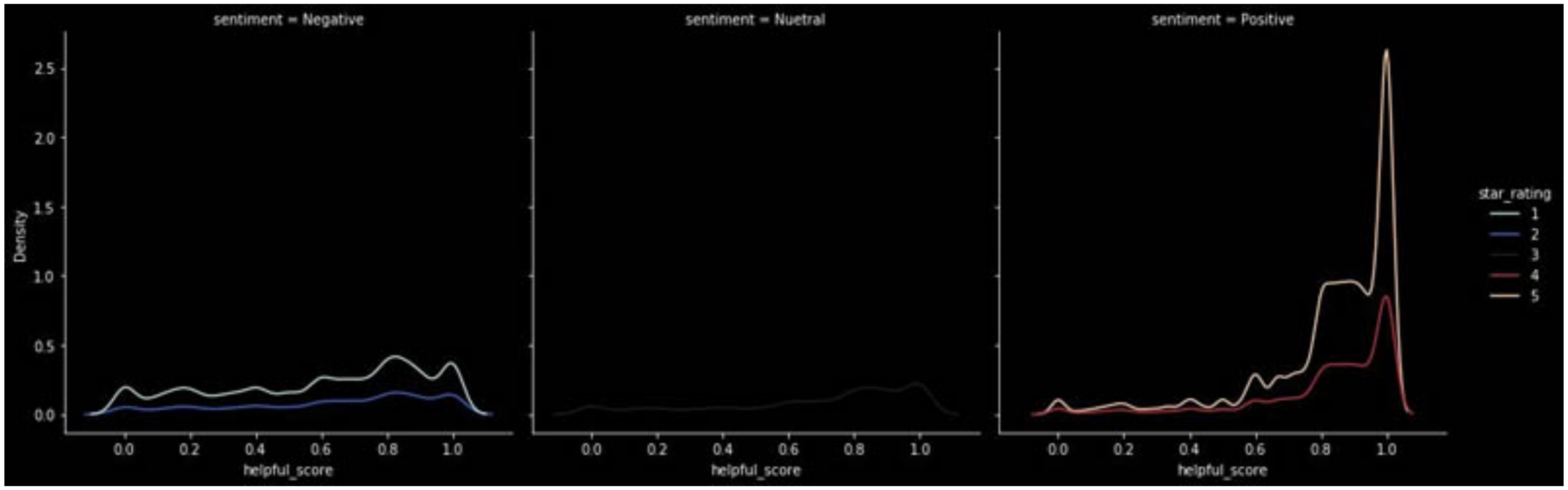
Data preparation

In this step, we download the electronics reviews from the Amazon Customer Reviews dataset. This dataset contains `star_rating` as well as `helpful_score` and `total_score` fields for each review.

We perform some feature engineering to calculate a helpful score for all reviews with at least five total votes:

```
df_reviews = df_reviews[df_reviews['total_votes'] >= 5]
df_reviews['helpful_score'] = df_reviews['helpful_votes'] / df_reviews['total_votes']
df_reviews['sentiment'] = pd.cut(df_reviews['star_rating'], bins=[0,2,3,6], labels=['Negative','Nuetral','Positi
df_reviews.describe()
```

The following visualization of the results shows that a helpful score threshold of `0.8` aligns well with high-rated products.



We use this as a target variable to create our binary classifier:

```
df_reviews['is_helpful'] = (df_reviews['helpful_score'] > 0.80)
```

We then split the data into training, test, and validation datasets. Transform the training data into the format required for the SageMaker Blazing Text algorithm and upload the text files to Amazon S3.

Run a SageMaker pipeline

In this step, we create a SageMaker pipeline to train and register our model with the deployment project we created.

1. Edit the notebook cell to update <<project_name>> with your project name, for example `ab-testing-pipeline`:

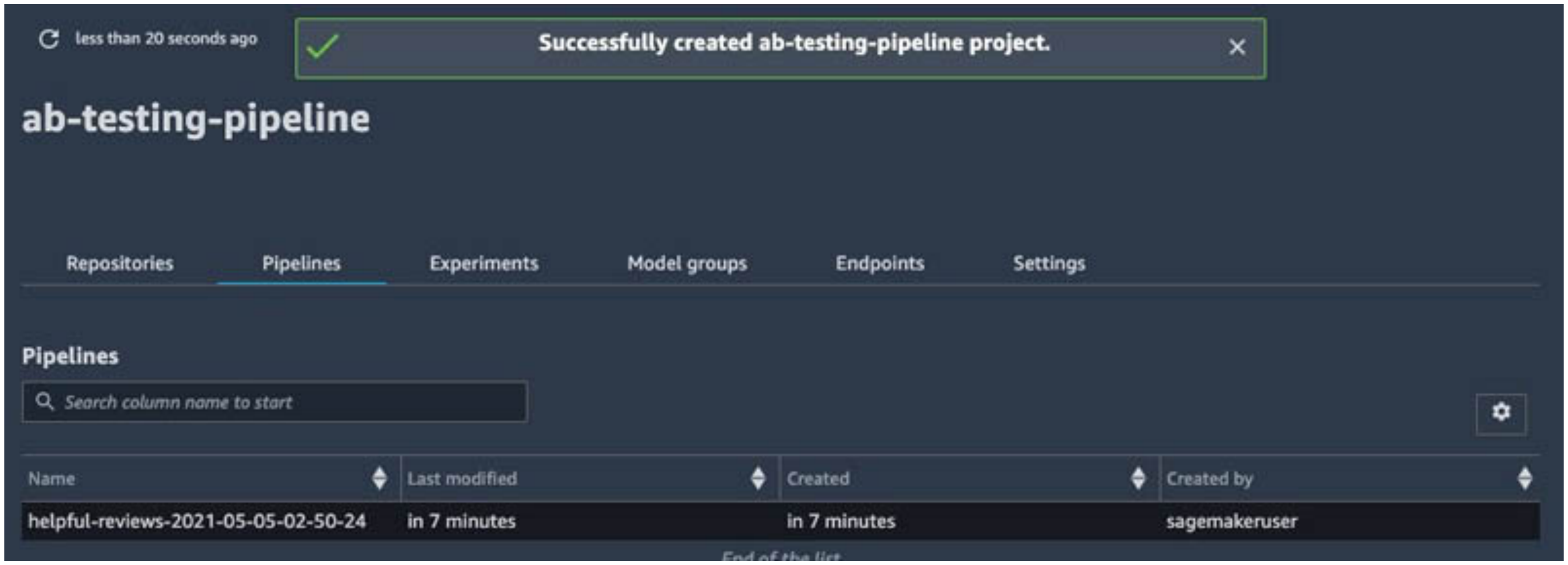
```
project_name = '<<project_name>>' # << Update with A/B testing deployment project
```

In the next series of cells, you define a SageMaker pipeline that has two steps: training the model and registering the model.

2. Continue running the cells and choose **Start Pipeline**.

The pipeline takes a few minutes to run.

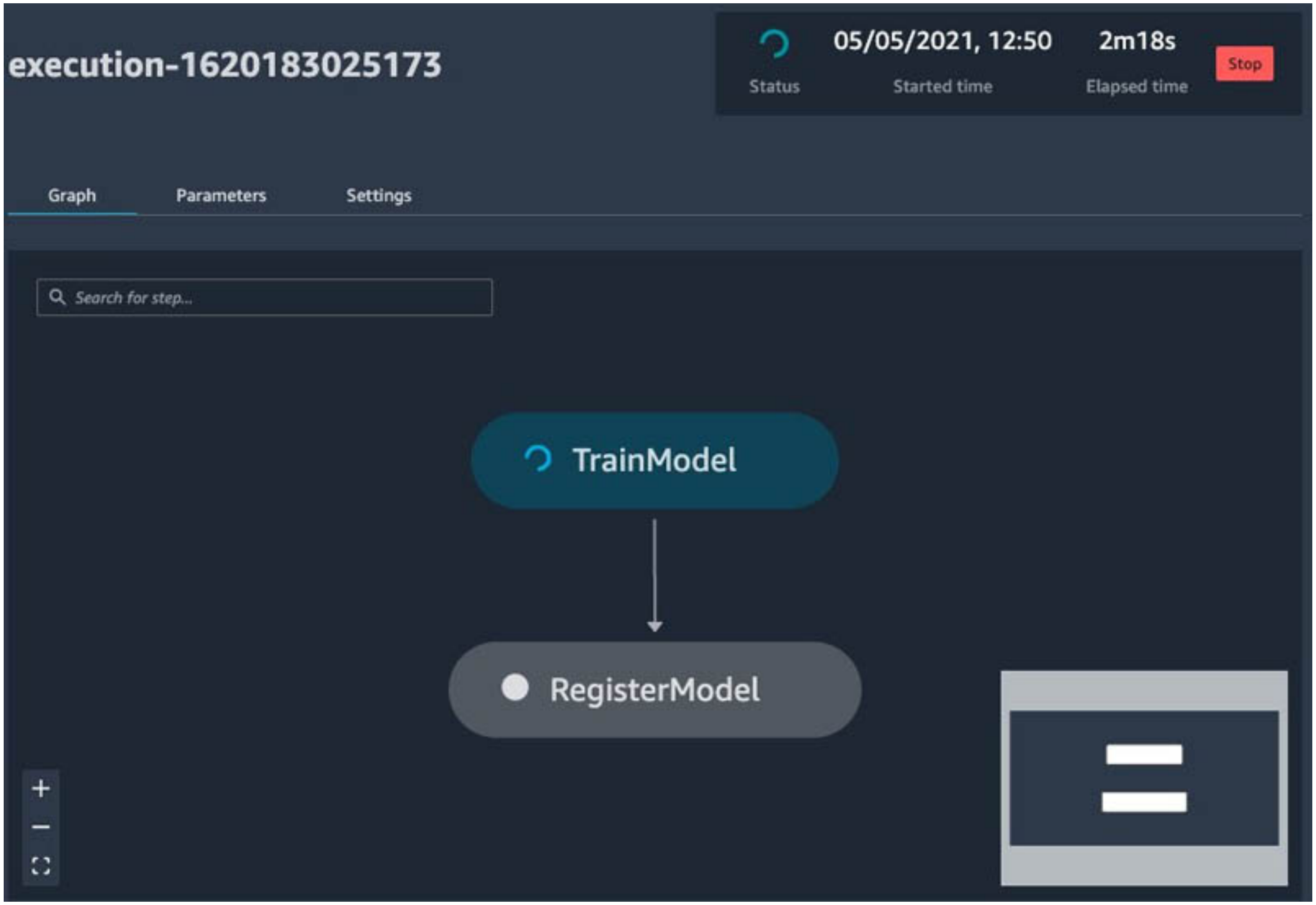
3. Navigate back to your project, which should now be created, and choose the **Pipelines**



Name	Last modified	Created	Created by
helpful-reviews-2021-05-05-02-50-24	in 7 minutes	in 7 minutes	sagemakeruser

4. Choose the pipeline to get the list of **Executions**.

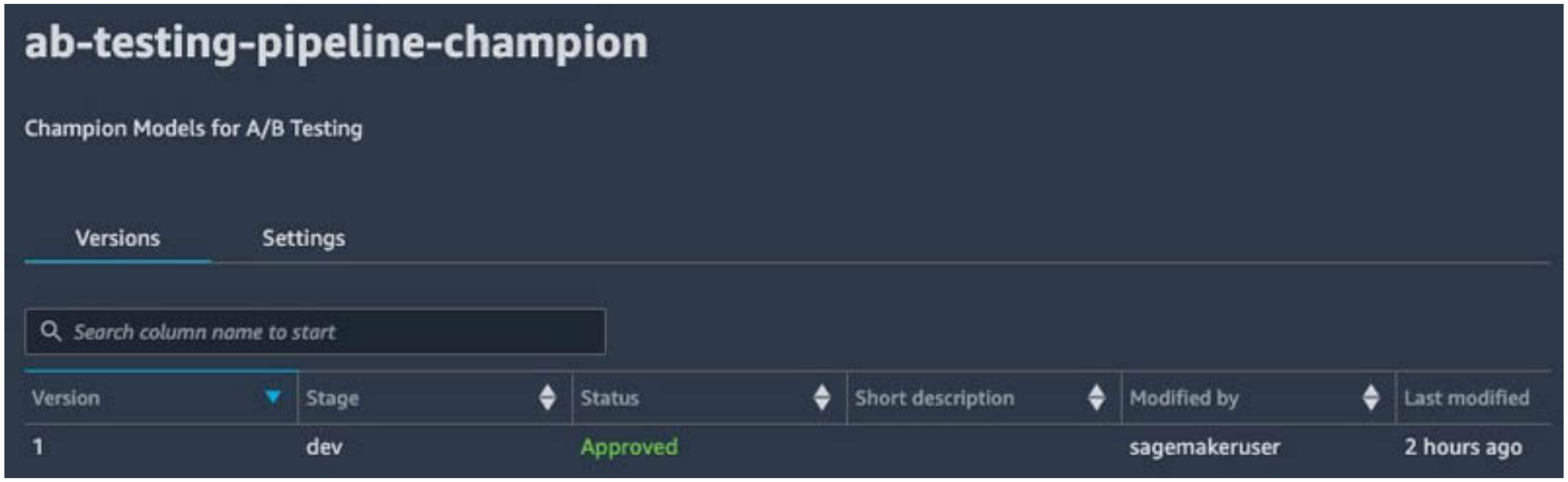
5. Choose the pipeline run to see the graph visualization that includes the `TrainModel` and `RegisterModel` steps.



6. Return to the notebook to approve the latest model package as our initial champion model:

```
champion_model_group = f"{project_name}-champion"

# Get the latest champion model package
packages = sm_client.list_model_packages(ModelPackageGroupName=champion_model_group,
                                           SortBy='CreationTime', SortOrder='Descending',
                                           MaxResults=1)['ModelPackageSummaryList']

# Approve the model
for package in packages:
    latest_model_package_arn = package['ModelPackageArn']
    model_package_version = latest_model_package_arn.split('/')[-1]
    if package['ModelApprovalStatus'] == 'PendingManualApproval':
        print(f"Approving Champion Version: {model_package_version}")
        model_package_update_response = sm_client.update_model_package(
            ModelPackageArn=latest_model_package_arn,
            ModelApprovalStatus="Approved",
        )
    else:
```

You can also manually approve models in Studio via the **Model Groups** tab in the project page.


ab-testing-pipeline-champion

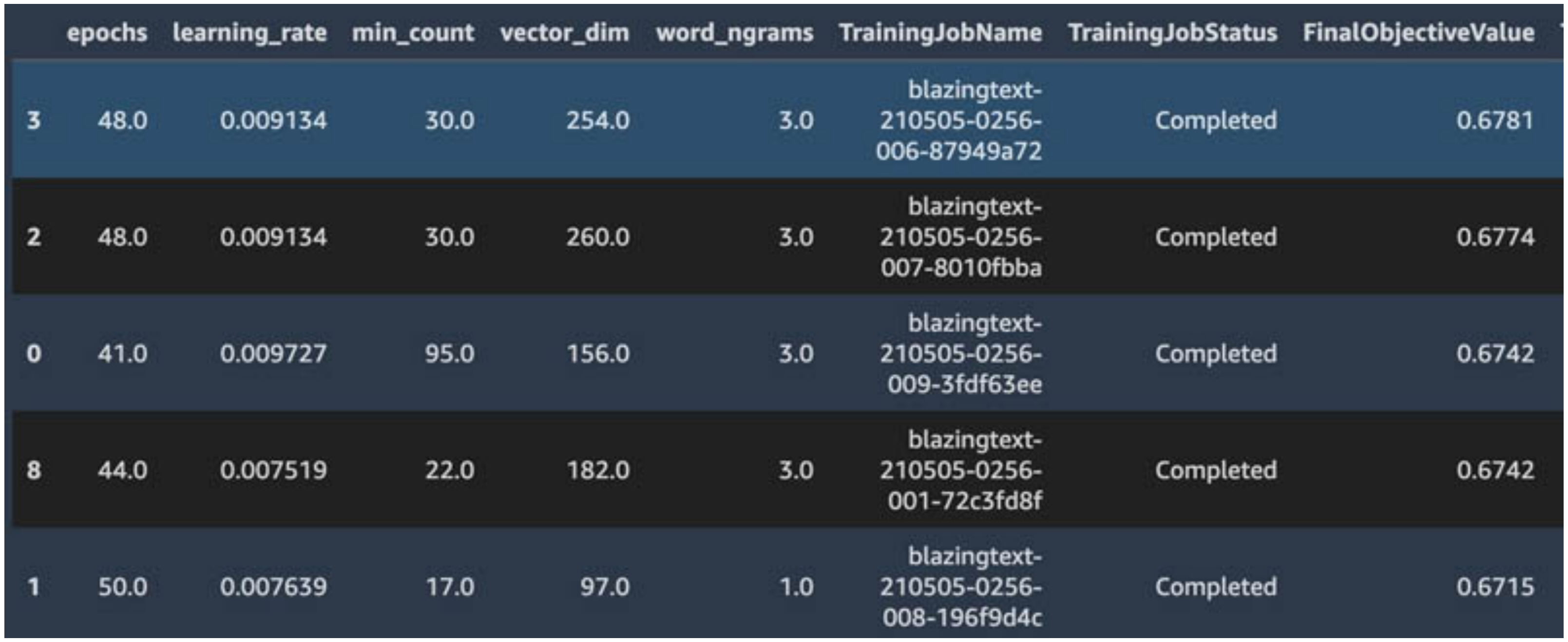
Champion Models for A/B Testing

Version	Stage	Status	Short description	Modified by	Last modified
1	dev	Approved		sagemakeruser	2 hours ago

Run a tuning job

In this next step of the notebook, we run a SageMaker tuning job to improve on this initial model for our A/B test. The notebook is configured to run a total of nine jobs with three in parallel. This process takes approximately 30 minutes to complete.

When this is complete, we can list these training jobs sorted by accuracy and see the hyperparameters identified in the best-performing training job.



	epochs	learning_rate	min_count	vector_dim	word_ngrams	TrainingJobName	TrainingJobStatus	FinalObjectiveValue
3	48.0	0.009134	30.0	254.0	3.0	blazingtext-210505-0256-006-87949a72	Completed	0.6781
2	48.0	0.009134	30.0	260.0	3.0	blazingtext-210505-0256-007-8010fbba	Completed	0.6774
0	41.0	0.009727	95.0	156.0	3.0	blazingtext-210505-0256-009-3fdf63ee	Completed	0.6742
8	44.0	0.007519	22.0	182.0	3.0	blazingtext-210505-0256-001-72c3fd8f	Completed	0.6742
1	50.0	0.007639	17.0	97.0	1.0	blazingtext-210505-0256-008-196f9d4c	Completed	0.6715

These metrics are also visible on the **Experiments** tab of the SageMaker project, where you can view and compare results.

If we're happy with the performance, we can register and approve this model in our challenger model group.

```
best_estimator = tuner.best_estimator()

challenger_model_group = f"{project_name}-challenger"

model_package = best_estimator.register(
    content_types=["text/plain"],
    response_types=["text/csv"],
    inference_instances=["ml.t2.large", "ml.m5.xlarge"],
    transform_instances=["ml.m5.xlarge"],
    model_package_group_name=challenger_model_group,
    approval_status="Approved"
)

model_package_version = model_package.model_package_arn.split('/')[-1]
print(f"Registered and Approved Challenger Version: {model_package_version}")
```

This triggers the A/B testing deployment pipeline, which creates a multi-variant SageMaker endpoint with the latest champion model and the challenger model you just approved. The production variant name is prefixed with `champion` or `challenger` and followed by the version number (for example, `Challenger1`).

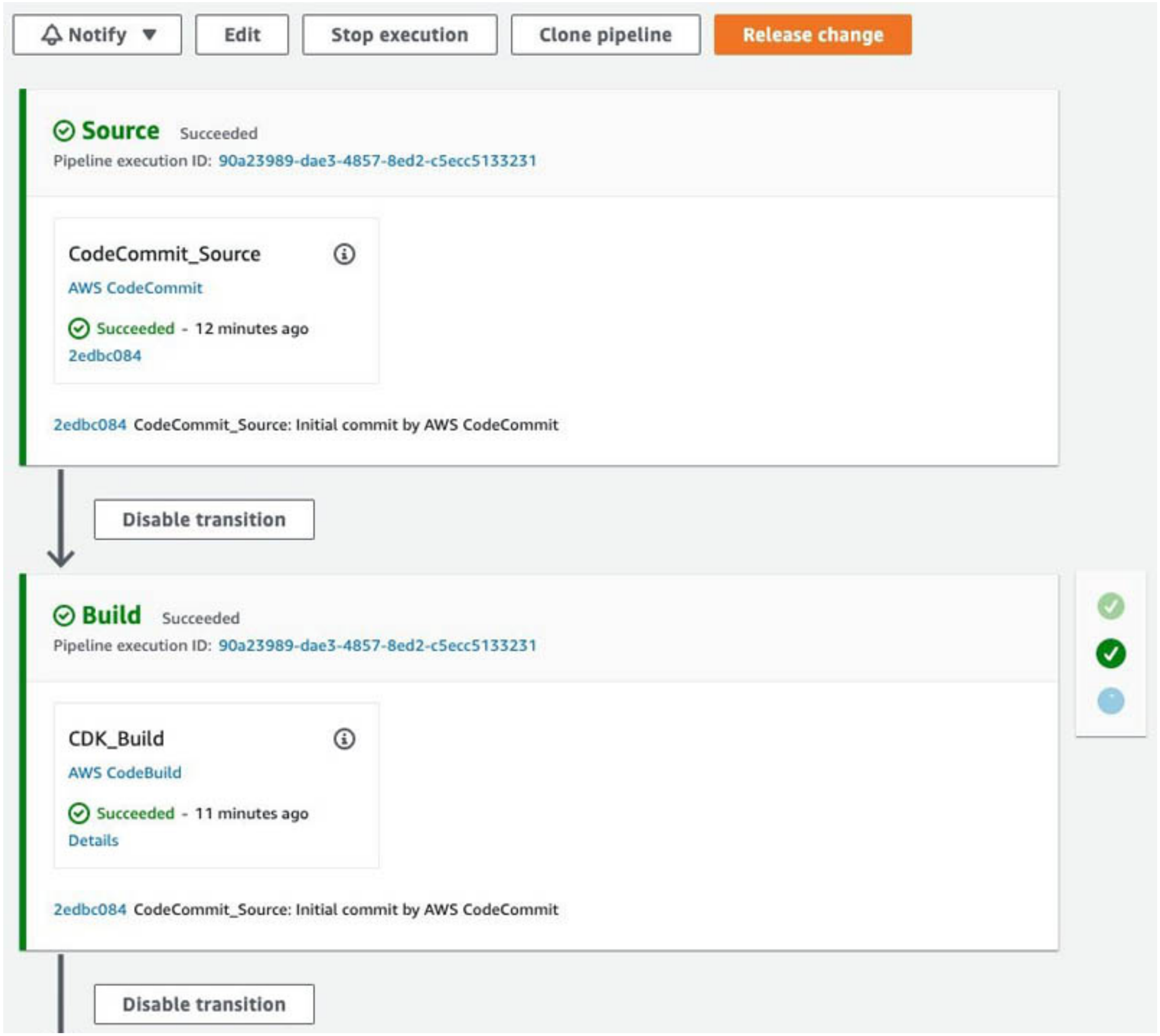
The deployment pipeline reads a JSON configuration to determine settings to provision the endpoint:

- **stage_name** – The stage suffix for the SageMaker endpoint (for example, `dev`)
- **instance_count** – The number of instances to deploy per variant
- **instance_type** – The type of instance to deploy per variant
- **challenger_variant_count** – The number of challenger models to deploy
- **strategy** – The algorithm strategy for selecting user model variants (`WeightedSampling`, `EpsilonGreedy`, `UCB1`, or `ThompsonSampling`)
- **epsilon** – The epsilon parameter used by the `EpsilonGreedy` strategy
- **warmup** – The number of invocations to warm up with `WeightedSampling` before applying the strategy

The CodeBuild stage in the deployment pipeline queries the SageMaker model registry for the latest N challenger models created after the top champion model. You also have the option of specifying the explicit `champion_variant_config` and `challenger_variant_config` model versions and `variant_name` as well as overriding the `instance_count` and `instance_type`, as demonstrated in the following example production config file:

```
{
    "stage_name": "prod",
    "strategy": "EpsilonGreedy",
    "warmup": 100,
    "epsilon": 0.1,
    "instance_count": 2,
    "instance_type": "ml.c5.large",
    "champion_variant_config": {
        "model_package_version": 1,
        "variant_name": "Champion",
        "instance_count": 3,
        "instance_type": "ml.m5.xlarge"
    },
    "challenger_variant_config": [
        {
            "model_package_version": 1,
            "variant_name": "Challenger1",
            "instance_type": "ml.c5.xlarge"
```

The CodeBuild stage of the pipeline uses AWS CDK to create the CloudFormation template.



Source Succeeded
Pipeline execution ID: 90a23989-dae3-4857-8ed2-c5ecc5133231

CodeCommit_Source
AWS CodeCommit
Succeeded - 12 minutes ago
2edbc084

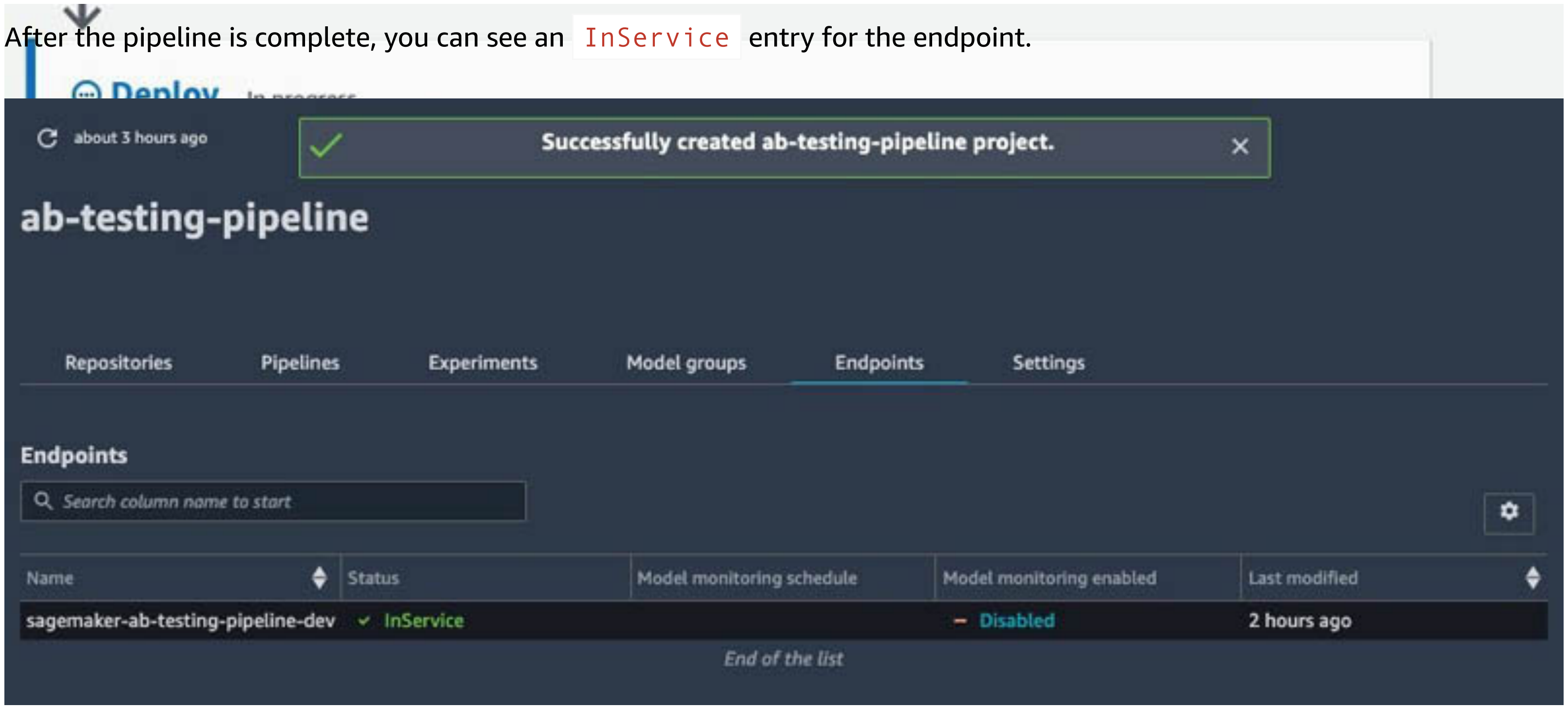
2edbc084 CodeCommit_Source: Initial commit by AWS CodeCommit

Build Succeeded
Pipeline execution ID: 90a23989-dae3-4857-8ed2-c5ecc5133231

CDK_Build
AWS CodeBuild
Succeeded - 11 minutes ago
Details

2edbc084 CodeCommit_Source: Initial commit by AWS CodeCommit

After the pipeline is complete, you can see an `InService` entry for the endpoint.



Successfully created ab-testing-pipeline project.

about 3 hours ago

ab-testing-pipeline

Endpoints

Name	Status	Model monitoring schedule	Model monitoring enabled	Last modified
sagemaker-ab-testing-pipeline-dev	✓ InService		– Disabled	2 hours ago

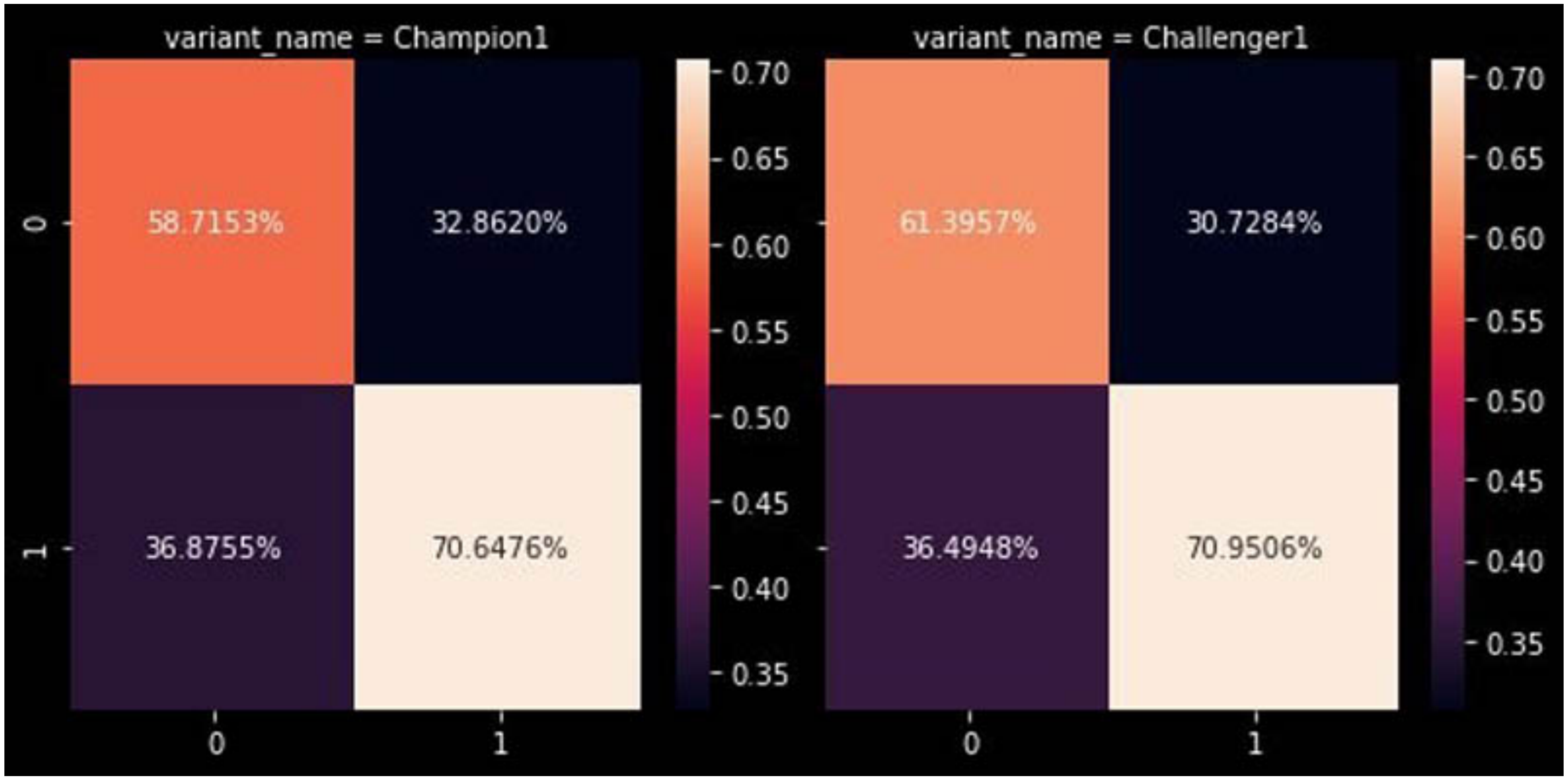
End of the list

Test the endpoint

With our new multi-variant endpoint in service, we can perform an offline evaluation with our test set and calculate an overall accuracy for each of these models (your results might vary slightly):

```
variant_name
Challenger1    0.667159
Champion1      0.653592
dtype: float64
```

Because this is a binary classifier for review helpfulness, we can use a confusion matrix to visualize the number of times our model correctly predicted the review as helpful (true positive) or not helpful (true negative), which shows slight improvement in both instances for our Challenger1 model.



Run the A/B testing simulation

In this step, we use the same test data to run an A/B testing simulation.

We first edit the notebook cell to update `rest_api` with the `ApiEndpoint` output from AWS CloudFormation:

```
rest_api = 'https://<<domain>>.execute-api.<<region>>.amazonaws.com/prod/' # << Update this with Rest API output
```

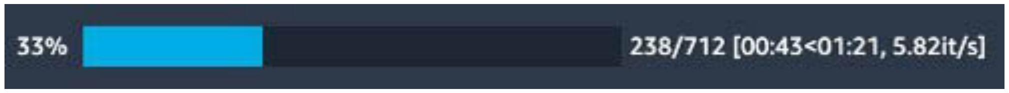
We define a few Python functions to interface with the API, which pass a `user_id` and `endpoint_name` to the invocation endpoint. We also provide a wrapper for the conversion and stats endpoints:

```
import json
import os
import requests

def api_invocation(user_id, text_array):
    payload = {
        "user_id": str(user_id),
        "endpoint_name": endpoint_name,
        "content_type": "application/json",
        "data": json.dumps({"instances" : text_array, "configuration": { "k": 1 }}),
    }
    rest_url = os.path.join(rest_api, 'invocation')
    r = requests.post(rest_url, data=json.dumps(payload))
    return r.json()

def api_conversion(payload):
    rest_url = os.path.join(rest_api, 'conversion')
    r = requests.post(rest_url, data=json.dumps(payload))
    return r.json()
```

The stats endpoint returns the variant metrics for a given `endpoint_name`, which are all zero to start our test:

```
{
 'endpoint_name': 'sagemaker-ab-testing-pipeline-dev',
 'variant_metrics': [
  {'endpoint_name': 'sagemaker-ab-testing-pipeline-dev',
   'variant_name': 'Champion1',
   'initial_variant_weight': 1.0,
   'invocation_count': 0,
   'conversion_count': 0,
   'reward_sum': 0.0},
  {'endpoint_name': 'sagemaker-ab-testing-pipeline-dev',
   'variant_name': 'Challenger1',
   'initial_variant_weight': 1.0,
   'invocation_count': 0,
   'conversion_count': 0,
   'reward_sum': 0.0}
 ],
 'strategy': 'ThompsonSampling',
 'epsilon': 0.1,
```

Next, we split our test review dataset into batches of 20 and invoke the API with a new user on each batch. The API allocates a variant for each user based on the bandit algorithm strategy, and returns this along with the strategy selected in the response. This process takes a few minutes to complete, as indicated by the progress bar.

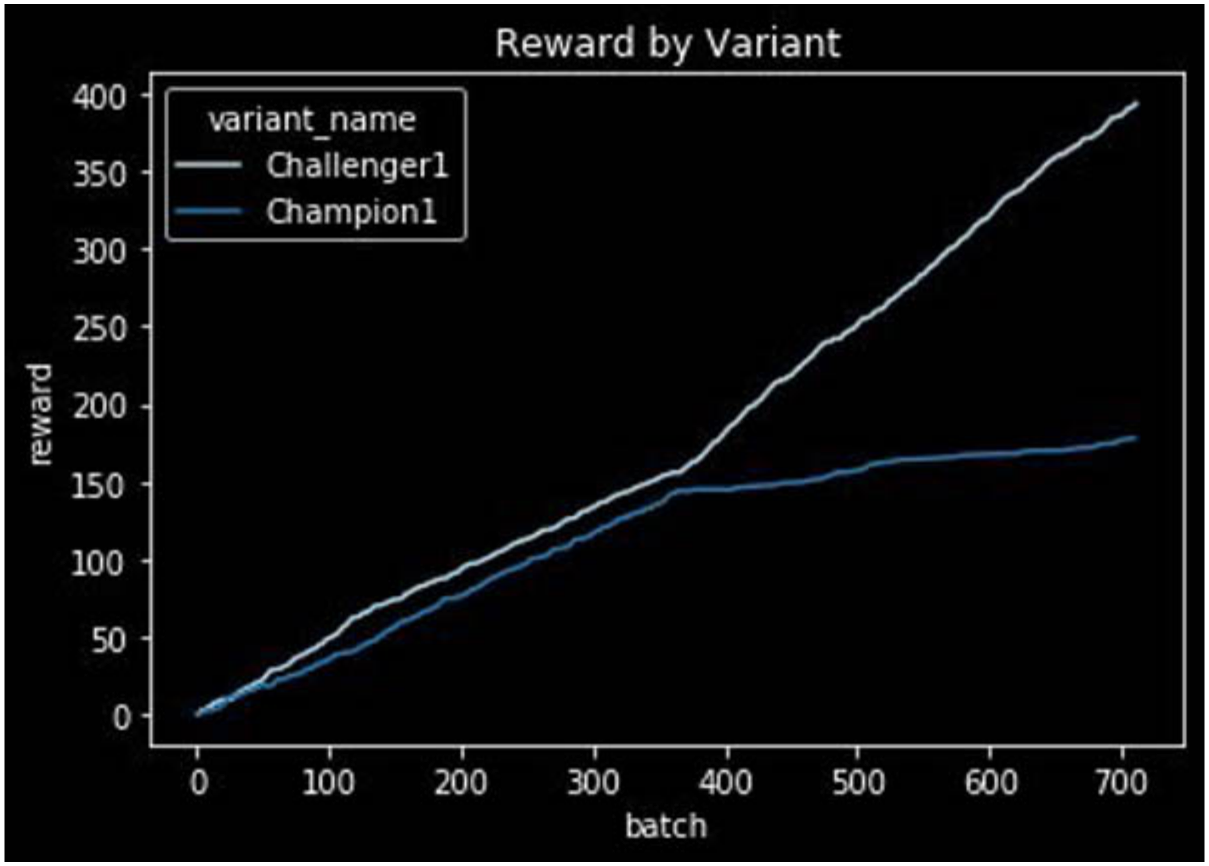


```
33% 238/712 [00:43<01:21, 5.82it/s]
```

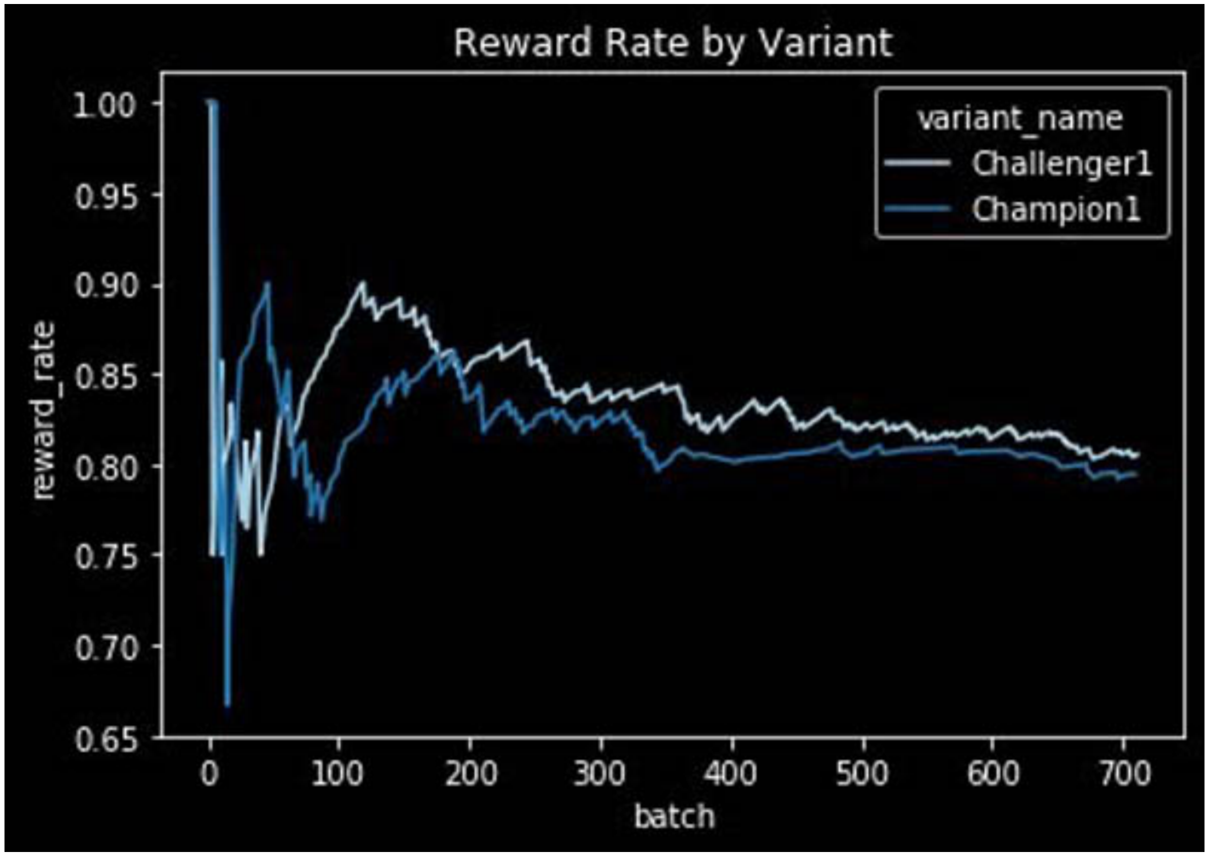
Our results show that for the first half of the test, the `WeightSampling` strategy is selected. This is because metrics are updated periodically via the Firehose delivery stream, and the bandit algorithms require reward feedback before they can suggest which variant to assign for a given user.

```
variant_name  strategy
Challenger1   ThompsonSampling    302
              WeightedSampling    186
Champion1     ThompsonSampling     47
              WeightedSampling    177
Name: is_helpful_prediction, dtype: int64
```

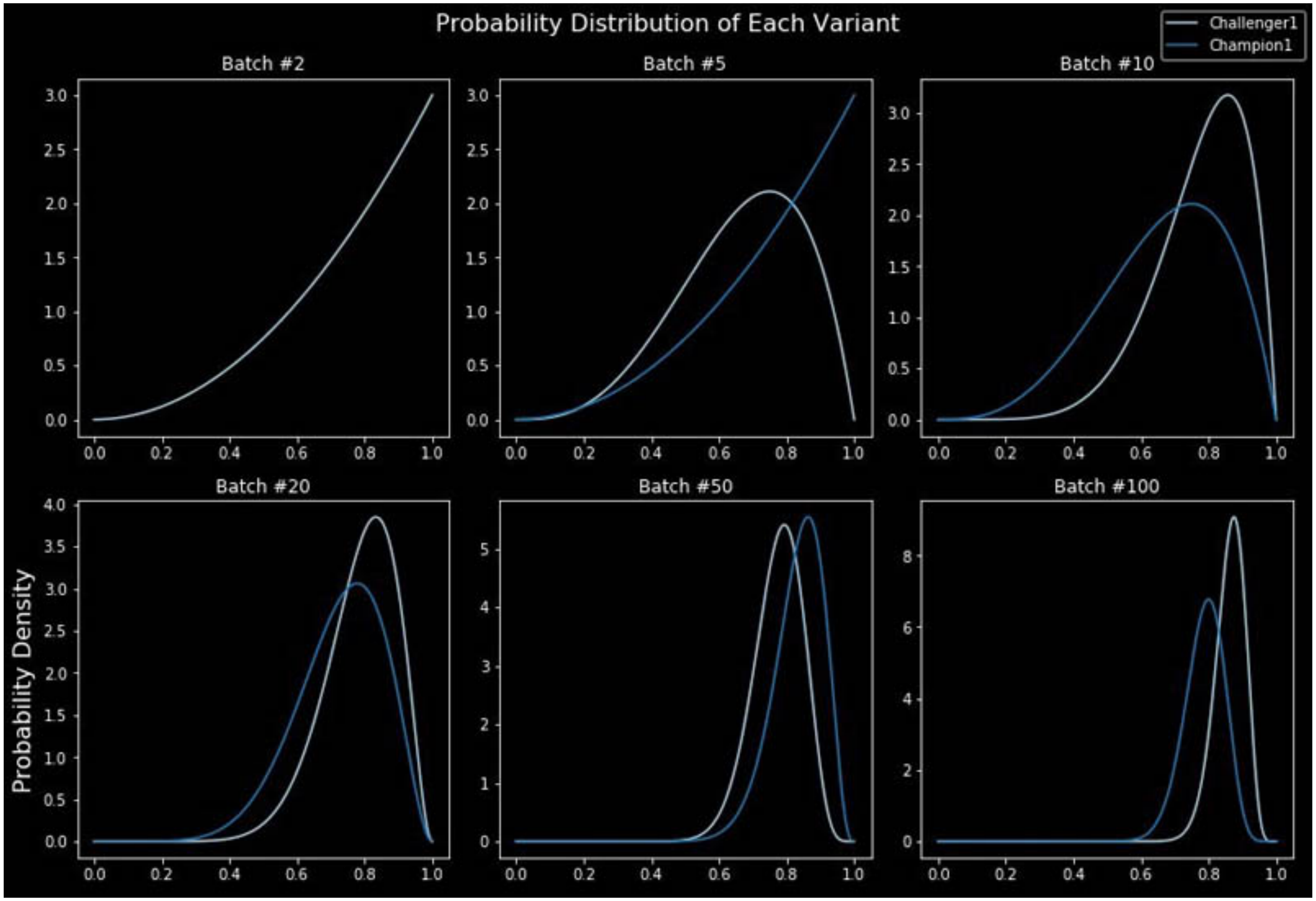
When we plot the cumulative reward for the variants `Champion1` and `Challenger1`, we can see that in the beginning they're both increasing at a steady rate.



But when our bandit algorithm starts to use the reward rate, we start to favor the `Challenger1` model variant because it's outperforming `Champion1`.



If we visualize the beta probability distribution over time with these batches, we can see the shape of the `Challenger1` distribution starts to become taller and skinner as it gets more helpful vs. not helpful reviews. When randomly sampling in these distributions with the Thompson sampling strategy, we're more likely to select the `Challenger1` variant as it continues to separate from the original `Champion1` model.



Determine the winning model

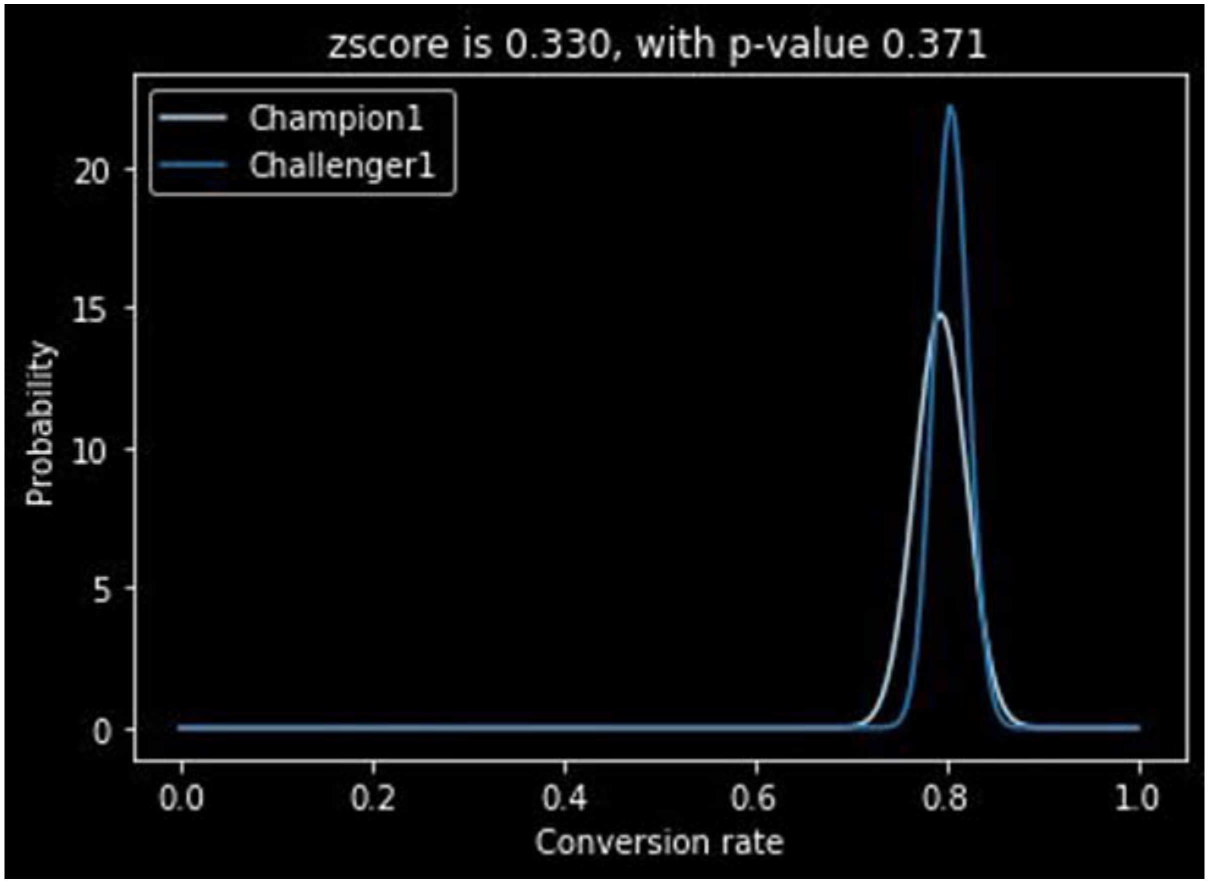
When setting up an A/B testing experiment, you need to do the following:

- Identify the business metric that you want to improve, such as such as click-through rate or conversion rate
- Identify the expected performance improvement for that metric (for example, 5%)

Multi-armed bandits provide the opportunity to determine the winner early when the following conditions are met:

- The experiment has received regular traffic
- The experiment has run for sufficient time to cancel out any periodicity (for example, 2 weeks)
- The new variant has exceeded the expected performance improvement.

In our simulation, we can calculate a conversion rate improvement for `Challenger1` over `Champion1`. Assuming a normal distribution, we evaluate whether this is a statistically significant result with a 95% confidence interval. In my case the results were too close to call, so the recommendation is to continue running this test.



Clean up

To clean up all the resources you provisioned in this example, complete the following steps in your terminal, using the AWS Command Line Interface (AWS CLI):

1. Delete the CloudFormation stack created to provision the SageMaker endpoint:

```
aws cloudformation delete-stack --stack-name sagemaker-<<project_name>>-deploy-<<stage_name>>
```

2. Empty the S3 bucket containing the artifacts output from the A/B testing deployment pipeline:

```
aws s3 rm --recursive s3://sagemaker-<<project_id>>-artifact-<<stage_name>>-<<region_name>>
```

3. Delete the project, which removes the CloudFormation stack that created the deployment pipeline:

```
aws sagemaker delete-project --project-name <<project_name>>
```

4. Delete the AWS Service Catalog project template:

```
cdk destroy ab-testing-service-catalog
```

5. Finally, delete the API and testing infrastructure:

```
cdk destroy ab-testing-api
```

Conclusion and next steps

In this post, you learned how to apply A/B testing to ML models. You used the AWS CDK to publish a custom MLOps template, and deployed a general purpose API and testing infrastructure to simulate A/B testing for recommending helpful reviews.

Because you assigned users to model variants using the Thompson sampling multi-armed bandit strategy, you reduced the volume of traffic sent to the poor-performing variant in the A/B test and reduced the time to call a winner.

As a next step, try using the A/B testing deployment pipeline for your own models in production. The source code is available on the GitHub repo.

About the Author

Julian Bright is an Sr. AI/ML Specialist Solutions Architect based out of Melbourne, Australia. Julian works as part of the global AWS machine learning team and is passionate about helping customers realise their AI and ML journey through MLOps. In his spare time he loves running around after his kids, playing soccer and getting outdoors.

Comments