INDEX

Sr. No.	Practical Aim	Signature
1.	Implementing advanced deep learning algorithms such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs) using Python libraries like TensorFlow or PyTorch.	
2.	Building a natural language processing (NLP) model for sentiment analysis or text classification.	
3.	Creating a chatbot using advanced techniques like transformer models.	
4.	Developing a recommendation system using collaborative filtering or deep learning approaches.	
5.	Implementing a computer vision project, such as object detection or image segmentation.	
6.	Training a generative adversarial network (GAN) for generating realistic images	
7.	Applying reinforcement learning algorithms to solve complex decision-making problems.	
8.	Utilizing transfer learning to improve model performance on limited datasets.	
9.	Building a deep learning model for time series forecasting or anomaly detection	
10.	Implementing a machine learning pipeline for automated feature engineering and model selection.	

PRACTICAL – 1

Aim: Implementing advanced deep learning algorithms such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) using popular Python libraries like TensorFlow or PyTorch

1. Convolutional Neural Network (CNN) for Image Classification

CNNs are widely used in computer vision tasks, such as image classification, object detection, and segmentation. They are particularly effective for processing grid-like data (e.g., images) due to their ability to capture spatial hierarchies.

We'll implement a CNN for image classification on the **MNIST** dataset, a collection of handwritten digits.

Step 1: Install Dependencies

Install the required libraries if you don't have them:

```
pip install tensorflow matplotlib numpy
```

Step 2: Load and Preprocess the Dataset

We'll use the **MNIST** dataset, which is available directly in TensorFlow. `import`

```
tensorflow as tf
```

```
from tensorflow.keras import layers, models import
numpy as np
import matplotlib.pyplot as plt

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
# Normalize the images to [0, 1] and reshape them to (28, 28, 1) x_train
= x_train.astype('float32') / 255.0 x_test = x_test.astype('float32') /
255.0
x_train = np.expand_dims(x_train, axis=-1)  # Add channel dimension
x_test = np.expand_dims(x_test, axis=-1)  # Add channel dimension
# One-hot encode the labels
y_train = tf.keras.utils.to_categorical(y_train, 10) y_test =
tf.keras.utils.to_categorical(y_test, 10)
```

Step 3: Define the CNN Model

We will create a simple CNN architecture with 2 convolutional layers, followed by a fully connected (dense) layer.

```
def build_cnn_model():     model
= models.Sequential()
# First convolutional layer
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))

    # Second convolutional layer
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

    # Flatten the output for the dense layer
model.add(layers.Flatten())
    # Fully connected layer
    model.add(layers.Dense(64, activation='relu'))

    # Output layer
    model.add(layers.Dense(10, activation='softmax'))  # 10 classes for MNIST digits
return model

# Build and compile the model cnn_model
= build_cnn_model()
cnn_model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
metrics=['accuracy'])
```

Step 4: Train the Model

Now, we'll train the CNN on the MNIST dataset.

```
# Train the CNN model cnn_model.fit(x_train, y_train, epochs=5, batch_size=64,
validation_split=0.1)
```

Step 5: Evaluate the Model

After training, we'll evaluate the model on the test set.

```
#
Evaluate the model on the test data
test_loss, test_acc = cnn_model.evaluate(x_test, y_test) print(f'Test
accuracy: {test_acc}')
```

Step 6: Visualize the Results

You can visualize the results by plotting the predictions made by the CNN.

```
# Make predictions predictions =
cnn_model.predict(x_test)

# Display the first 5 images and their predicted labels for
i in range(5):     plt.imshow(x_test[i].reshape(28, 28),
cmap='gray')     plt.title(f"Predicted:
```

```
{np.argmax(predictions[i])}, Actual:
{np.argmax(y_test[i])}")     plt.show()
```

2. Recurrent Neural Network (RNN) for Sequence Prediction

RNNs are well-suited for sequential data like time-series, text, or audio. They maintain hidden states over time, making them effective for sequence modeling.

Let's implement a simple RNN using TensorFlow to predict the next word in a sequence.

Step 1: Install Dependencies

If you don't have the required libraries yet, install them:

```
pip install tensorflow numpy
```

Step 2: Prepare the Dataset

We'll use the **IMDB dataset** (a movie review dataset) for text classification. We'll build an RNN to predict the sentiment of a movie review (positive or negative).

```
# Load IMDB dataset for sentiment analysis from tensorflow.keras.datasets
import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Set maximum number of words to consider and maximum sequence length
max_features = 10000  # Only consider the top 10,000 words maxlen = 500
# Maximum length of the review

# Load and preprocess the dataset
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
# Pad sequences to ensure they have the same length x_train
= pad_sequences(x_train, maxlen=maxlen) x_test
= pad_sequences(x_test, maxlen=maxlen)
```

Step 3: Define the RNN Model

We will define an RNN with **LSTM** layers. LSTM (Long Short-Term Memory) is an advanced type of RNN that solves the vanishing gradient problem, making it more effective for long sequences. `def build_rnn_model():  model = models.Sequential()`

```
    # Embedding layer to learn word representations
    model.add(layers.Embedding(input_dim=max_features, output_dim=128,
input_length=maxlen))

    # LSTM layer
    model.add(layers.LSTM(128))
# Output layer (sigmoid for
binary classification)
```

```
model.add(layers.Dense(1,
activation='sigmoid'))
          return
model

# Build and compile the RNN model rnn_model
= build_rnn_model()
rnn_model.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
```

Step 4: Train the Model

We will train the RNN model on the IMDB dataset.

```
# Train the RNN model
rnn_model.fit(x_train, y_train, epochs=5, batch_size=64, validation_data=(x_test,
y_test))
```

Step 5: Evaluate the Model

After training, we can evaluate the model's performance on the test data.

```
# Evaluate the model on the test data
test_loss, test_acc = rnn_model.evaluate(x_test, y_test) print(f'Test
accuracy: {test_acc}')
```

Step 6: Make Predictions

Once the model is trained, we can use it to make predictions on new reviews.

```
# Predict sentiment for the first review in the test set predictions =
rnn_model.predict(x_test[:5])

# Print predictions for i, prediction in enumerate(predictions):     sentiment
= 'positive' if prediction >= 0.5 else 'negative'     print(f"Review {i+1}:
{sentiment} (probability: {prediction[0]:.2f})")
```

Conclusion

In this we've demonstrated how to implement advanced deep learning algorithms using **TensorFlow**:

1. **CNN**: We used a convolutional neural network to classify handwritten digits from the MNIST dataset. CNNs are powerful for image recognition tasks, and they work by learning spatial hierarchies through convolution and pooling operations.
2. **RNN**: We used a recurrent neural network (RNN) with LSTM units to classify sentiment from movie reviews in the IMDB dataset. RNNs are ideal for sequence data because they maintain hidden states across time steps, enabling them to capture dependencies in the data.

PRACTICAL – 2

Aim: Building a Natural Language Processing (NLP) model for sentiment analysis or text classification

Sentiment analysis is a common Natural Language Processing (NLP) task that involves determining the sentiment or emotional tone of a text. This can be categorized as positive, negative, or neutral. Let's walk through a basic sentiment analysis example using Python and a popular NLP library, Huggingface Transformers.

We will use the **IMDb movie reviews dataset** to classify the sentiment of movie reviews as **positive** or **negative**. This dataset contains movie reviews labeled as 1 (positive) or 0 (negative), making it a binary classification task.

Steps for Sentiment Analysis

1. Install the required libraries:

```
pip install transformers datasets torch
```

- `transformers`: For using pre-trained models like BERT.
- `datasets`: A Huggingface library to load datasets like IMDb. `torch`: For deep learning with PyTorch.

2. Import the libraries:

```
from datasets import load_dataset
from transformers import BertTokenizer, BertForSequenceClassification, Trainer,
TrainingArguments import torch
```

3. Load the IMDb dataset:

The `datasets` library from Huggingface makes it easy to load popular NLP datasets.

```
# Load the IMDb dataset dataset =
load_dataset('imdb')
print(dataset)
```

This loads the IMDb dataset and gives you the following splits:

- `train`: Training data
- `test`: Test data

Each review is a string of text, and each label is a sentiment (0 for negative, 1 for positive).

4. Preprocess the data:

BERT requires tokenization, where the text is converted into token IDs that BERT can understand.

```
# Load the BERT tokenizer tokenizer =
BertTokenizer.from_pretrained('bert-base-uncased')
# Tokenization function def tokenize_function(examples):     return
tokenizer(examples['text'], padding="max_length", truncation=True)
# Apply tokenization to both the train and test data train_data =
dataset['train'].map(tokenize_function, batched=True) test_data =
dataset['test'].map(tokenize_function, batched=True)

# Set format for PyTorch
train_data.set_format(type='torch', columns=['input_ids', 'attention_mask',
'label']) test_data.set_format(type='torch', columns=['input_ids', 'attention_mask',
'label'])
```

Here, we:

- Load a pre-trained BERT tokenizer (`bert-base-uncased`), which converts text into tokens that the BERT model can understand.
- Define a `tokenize_function` that tokenizes the text and applies padding and truncation to ensure all sequences have the same length.
- Apply the tokenization function to both the training and testing datasets.
- Format the data into PyTorch tensors (`input_ids`, `attention_mask`, and `label`).

5. Load the pre-trained BERT model:

We now load the BERT model pre-trained on a large corpus of text and fine-tune it for sentiment analysis.

```
# Load the pre-trained BERT model for sequence classification (sentiment analysis)
model =     BertForSequenceClassification.from_pretrained('bert-base-uncased',
num_labels=2)
```

`num_labels=2` indicates that the model will predict two possible classes (positive or negative).

6. Define Training Arguments:

The `TrainingArguments` class is used to specify how the model should be trained.

```
# Set up training arguments training_args
= TrainingArguments(     output_dir='./results',          # Output directory
for model checkpoints
evaluation_strategy="epoch",     # Evaluate the model every epoch
learning_rate=2e-5,              # Learning rate for optimization
per_device_train_batch_size=16,  # Batch size for training
per_device_eval_batch_size=64,   # Batch size for evaluation     num_train_epochs=3,
# Number of training epochs     weight_decay=0.01,        # L2 regularization
)
```

```
# Set up the Trainer trainer
= Trainer(
    model=model,                        # The model to train
args=training_args,                 # Training arguments
train_dataset=train_data,           # Training dataset     eval_dataset=test_data,
# Evaluation dataset )
```

7. Train the Model:

Now we can start the training process.

```
# Train the model trainer.train()
```

The model will train for 3 epochs (as specified), during which it learns to classify the sentiment of reviews.

8. Evaluate the Model:

Once training is complete, we can evaluate how well the model performs on the test dataset.

```
# Evaluate the model results =
trainer.evaluate()
print(results)
```

The evaluation will return several metrics, including accuracy, precision, recall, and F1-score.

9. Predict Sentiment of New Reviews:

Now, let's make predictions on a new set of text data (movie reviews).

```
# Predict sentiment for a new review  text = "I love this movie! It was
amazing and the acting was superb." inputs = tokenizer(text,
return_tensors="pt", padding=True, truncation=True, max_length=512) with
torch.no_grad():     logits = model(**inputs).logits     prediction =
torch.argmax(logits, dim=-1).item()

# Convert the prediction to a sentiment label sentiment =
"Positive" if prediction == 1 else "Negative" print(f"Sentiment:
{sentiment}")
```

This code takes a new review (`text`), tokenizes it using the same tokenizer, and then feeds it into the trained BERT model to get the predicted sentiment.

Conclusion:

In this example, we've used Huggingface's Transformers library to build a sentiment analysis model using the BERT model. The key steps include loading and tokenizing data, fine-tuning a pre-trained model, and evaluating its performance. This approach can be applied to many text classification tasks by choosing different datasets and pre-trained models.

Overview of the Steps:

1. **Data Loading** 2. **Data Preprocessing** 3. **Feature Extraction** 4. **Model Training**

5. **Model Evaluation** 6. **Prediction**

We'll start by building a sentiment analysis model using Logistic Regression (Traditional Machine Learning model), and then use a deep learning-based model (BERT) for more advanced performance.

Step 1: Data Loading

We'll use the **IMDb dataset** for this example. The dataset contains movie reviews labeled as positive or negative.

```
from datasets import load_dataset

# Load IMDb dataset  dataset =
load_dataset('imdb')

# Inspect the data print(dataset)
```

- The `train` set consists of 25,000 movie reviews, labeled as positive (1) or negative (0).
- The `test` set is similarly structured.

Step 2: Data Preprocessing

We need to clean and tokenize the text before feeding it to the model. For traditional machine learning models, we can use **Bag of Words (BoW)** or **TF-IDF**. For deep learning models (like BERT), tokenization is handled by the model itself.

Preprocessing for Logistic Regression (Traditional Machine Learning):

We'll use TF-IDF to transform text into numerical features. `from`

```
sklearn.feature_extraction.text import TfidfVectorizer from

sklearn.model_selection import train_test_split

# Use the 'text' column for the review and 'label' for sentiment train_data
= dataset['train'] X = train_data['text'] y = train_data['label']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Use TF-IDF for feature extraction
tfidf = TfidfVectorizer(stop_words='english', max_features=5000)
```

```
X_train_tfidf = tfidf.fit_transform(X_train)
X_test_tfidf = tfidf.transform(X_test)
```

- **TF-IDF (Term Frequency-Inverse Document Frequency)** represents words based on their frequency in a document while considering the frequency across all documents, allowing the model to focus on important words.

Preprocessing for BERT (Deep Learning Model): BERT tokenization requires the Huggingface

`transformers` library, which tokenizes text into subwords and converts them into IDs that the model can understand.

```
from transformers import BertTokenizer

# Load the BERT tokenizer  tokenizer = BertTokenizer.from_pretrained('bert-base-
uncased')
# Tokenize the text def tokenize_function(examples):      return
tokenizer(examples['text'], padding=True, truncation=True,
max_length=512)

# Apply tokenization to both train and test datasets  train_data =
dataset['train'].map(tokenize_function, batched=True) test_data =
dataset['test'].map(tokenize_function, batched=True)

# Convert datasets to PyTorch format
train_data.set_format(type='torch', columns=['input_ids', 'attention_mask',
'label'])  test_data.set_format(type='torch', columns=['input_ids',
'attention_mask', 'label'])
```

- **Tokenization**: Converts text into numerical token IDs. BERT uses subwords, meaning words are often broken into smaller pieces for more accurate representations.

Step 3: Feature Extraction (TF-IDF vs. BERT)

- **TF-IDF** (Traditional Machine Learning Approach) creates a sparse matrix representing the text. **BERT** (Deep Learning Approach) uses embeddings that capture contextual meaning in text.

For traditional models, we use **TF-IDF**. For BERT, we directly use the tokenized inputs.

Step 4: Model Training

Logistic Regression (Traditional Machine Learning Approach):

We'll train a Logistic Regression classifier on the TF-IDF features.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
# Initialize Logistic Regression model lr_model
= LogisticRegression(max_iter=100)

# Train the model  lr_model.fit(X_train_tfidf,
y_train)

# Evaluate the model
y_pred = lr_model.predict(X_test_tfidf) print(classification_report(y_test,
y_pred))
```

Here we:

- Train a **Logistic Regression** model on the TF-IDF features.
- Evaluate the model using **classification metrics** (precision, recall, F1-score).

BERT (Deep Learning Approach):

For BERT, we'll use the `Trainer` API from the **Huggingface Transformers** library to fine-tune a pretrained BERT model. `from transformers import BertForSequenceClassification, Trainer, TrainingArguments`

```
# Load the pre-trained BERT model
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
num_labels=2)

# Define training arguments training_args
= TrainingArguments(     output_dir='./results',          # Output
directory for checkpoints
evaluation_strategy="epoch",     # Evaluate after each epoch
learning_rate=2e-5,              # Learning rate
per_device_train_batch_size=16,  # Training batch size
per_device_eval_batch_size=64,   # Evaluation batch size     num_train_epochs=3,
# Number of epochs     weight_decay=0.01,               # Regularization
)

# Initialize Trainer trainer =
Trainer(     model=model,
args=training_args,
train_dataset=train_data,
eval_dataset=test_data,
)

# Train the model trainer.train()

# Evaluate the model results =
trainer.evaluate()
print(results)
```

- **BERT** is a deep learning model trained on large amounts of text data. We fine-tune it for sentiment analysis using the IMDb dataset.

Step 5: Model Evaluation

For both models (Logistic Regression and BERT), we evaluate performance using metrics like **accuracy**, **precision**, **recall**, and **F1-score**.

For **Logistic Regression**, we use `classification_report` from **scikit-learn**. For **BERT**, we use the `Trainer`'s built-in evaluation functionality.

Step 6: Prediction

Once the model is trained, we can make predictions on new reviews:

Logistic Regression (Traditional Model): #

```
Predict sentiment for a new review

new_review = ["This movie was fantastic! I loved the acting and the plot."]
new_review_tfidf = tfidf.transform(new_review)
predicted_sentiment = lr_model.predict(new_review_tfidf)

sentiment = "Positive" if predicted_sentiment == 1 else "Negative"

print(f"Sentiment: {sentiment}")
```

BERT (Deep Learning Model):

```
# Tokenize the new review
inputs = tokenizer("This movie was fantastic! I loved the acting and the plot.",
return_tensors="pt", padding=True, truncation=True, max_length=512)
# Predict sentiment using the BERT model with
torch.no_grad():     logits =
model(**inputs).logits
    predicted_class = torch.argmax(logits, dim=-1).item()

sentiment = "Positive" if predicted_class == 1 else "Negative" print(f"Sentiment:
{sentiment}")
```

Conclusion:

- **Traditional Machine Learning** (e.g., Logistic Regression with TF-IDF) is faster and easier to implement but may not capture deep contextual information in text.
- **Deep Learning Models** (e.g., BERT) are more powerful and capture complex relationships and context in text, but they require more computational resources and time for training.

Both approaches are effective for sentiment analysis, but the choice between them depends on the specific use case, available resources, and the complexity of the task.

M.Sc. IT. SEM 3 **Advance Artificial Intelligence**

PRACTICAL – 3

Aim: Creating a chatbot using advanced techniques like the Transformer models.

Key Steps to Build a Transformer-Based Chatbot:

1. **Choose a Pre-trained Transformer Model**: You can use models like **GPT**, **DialoGPT**, or **BERT-based models** fine-tuned for conversational tasks.
2. **Set Up the Development Environment**: Install the required libraries.
3. **Data Preprocessing**: Although pre-trained models are already well-suited for many tasks, data can be fine-tuned for better domain-specific performance.
4. **Build the Chatbot Interface**: Set up the chatbot interface to communicate with the model.
5. **Deploy the Model**: You can run the model locally or deploy it on a cloud platform.

Step 1: Install Required Libraries

First, we need to install the **Huggingface Transformers** library or the **OpenAI GPT** API. Here's how to set up each.

Option 1: Huggingface (DialoGPT)

```
pip install transformers torch
```

Option 2: OpenAI API (for GPT-3/4)

For GPT-3/4 (via OpenAI's API), you first need an API key from OpenAI. `pip install`

```
openai
```

Step 2: Build a Simple Chatbot with DialoGPT (Huggingface)

DialoGPT is a variant of GPT-2 fine-tuned for conversation. We can use it to build an interactive chatbot.

1. Import the Libraries `from transformers import AutoModelForCausalLM,`

```
AutoTokenizer import
```

```
torch
```

2. Load the Pre-trained Model and Tokenizer

```
# Load DialoGPT model and tokenizer model_name
= "microsoft/DialoGPT-medium" model =
AutoModelForCausalLM.from_pretrained(model_name) tokenizer =
AutoTokenizer.from_pretrained(model_name)
```

- **DialoGPT-medium** is a smaller version of the model. You can also try **DialoGPT-large** if you need more power.

3. Create a Chat Function

```
def chat_with_bot(input_text, chat_history=None):
    # Encode the new user input, add the eos_token and return a tensor in Pytorch
new_user_input_ids = tokenizer.encode(input_text + tokenizer.eos_token,
return_tensors='pt')

    # Append the new user input tokens to the chat history (if exists)
bot_input_ids = new_user_input_ids if chat_history is None else
torch.cat([chat_history, new_user_input_ids], dim=-1)

    # Generate a response from the model
    chat_history = model.generate(bot_input_ids, max_length=1000,
pad_token_id=tokenizer.eos_token_id, no_repeat_ngram_size=3, top_p=0.92,
temperature=0.75)

    # Decode the generated response
    bot_output = tokenizer.decode(chat_history[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
          return bot_output,
chat_history
```

- **chat_with_bot()**: This function takes the user input and returns the model's response while maintaining the conversation history.
- **`max_length`**: Maximum number of tokens for the generated output.
- **`top_p` and `temperature`**: Control the creativity of the response. Lower values (e.g., 0.75) make the model more deterministic. **4. Interact with the Chatbot**

```
chat_history = None   while
True:     user_input =
input("You: ")
if user_input.lower() == "quit":         break
    bot_response, chat_history = chat_with_bot(user_input, chat_history)
print(f"Bot: {bot_response}")
```

This simple loop allows the user to interact with the chatbot in a conversational manner. Type **"quit"** to end the conversation.

Step 3: Build a Chatbot with GPT-3/4 (OpenAI API)

If you prefer to use **GPT-3/4**, which is very advanced and capable of handling more complex conversations, here's how you can do it using OpenAI's API.

1. Set Up OpenAI API Key

First, set your OpenAI API key. If you haven't gotten one, sign up at OpenAI's platform.

```
import openai


# Set up the OpenAI API key

openai.api_key = 'your-api-key-here'
```

2. Create a Function for Chatbot Interaction

```
def chat_with_gpt3(input_text):
response = openai.Completion.create(
        engine="text-davinci-003",  # You can use "gpt-3.5-turbo" or "gpt-4"
depending on your API access          prompt=input_text,          max_tokens=150,
temperature=0.7,          top_p=1.0,          frequency_penalty=0.0,
presence_penalty=0.0,          stop=["\n"]
    )
    return response.choices[0].text.strip()
```

- **engine**: Choose the model you want to use (e.g., `davinci`, `gpt-3.5-turbo`, `gpt-4`).
- **temperature**: Controls the randomness of the response (0.7 is a good middle ground).

3. Chat with GPT-3

```
while True:     user_input =
input("You: ")     if
user_input.lower() == "quit":
break
    bot_response = chat_with_gpt3(user_input)
print(f"Bot: {bot_response}")
```

Step 4: Improve the Chatbot with Memory (Optional) Example

(Huggingface DialoGPT with memory): `def`

```
chat_with_memory(input_text, chat_history=None):

    # Encode the user input and concatenate with previous chat history
new_user_input_ids = tokenizer.encode(input_text + tokenizer.eos_token,
return_tensors='pt')
    bot_input_ids = new_user_input_ids if chat_history is None else
```

```
torch.cat([chat_history, new_user_input_ids], dim=-1)
    # Generate response
    chat_history = model.generate(bot_input_ids, max_length=1000,
pad_token_id=tokenizer.eos_token_id, no_repeat_ngram_size=3, top_p=0.92,
temperature=0.75)
    bot_output = tokenizer.decode(chat_history[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
         return bot_output,
chat_history
```

Step 5: Deploy the Chatbot (Optional)

After developing your chatbot, you can deploy it to various platforms:

1. **Web Deployment**: Use **Flask** or **FastAPI** to build a web API for your chatbot.
2. **Messaging Platforms**: Integrate the chatbot into platforms like **Slack**, **Discord**, or **Telegram** using their respective bot APIs.

Conclusion

By leveraging pre-trained Transformer models like **DialoGPT** or **GPT-3/4**, you can quickly build a powerful and scalable chatbot. These models can understand context, generate human-like responses, and be fine-tuned for specific tasks or domains. Whether you use **Huggingface Transformers** or the **OpenAI API**, both approaches allow you to build state-of-the-art conversational agents.

M.Sc. IT. SEM 3 **Advance Artificial Intelligence**

PRACTICAL – 4

Aim: Developing a recommendation system using collaborative filtering or deep learning approaches.

1. **Collaborative Filtering** (a traditional method)
2. **Deep Learning Approaches** (using neural networks)

1. Collaborative Filtering

Collaborative filtering is based on the idea that users who agreed in the past will agree in the future. It predicts a user's preferences based on the preferences of other similar users.

There are two types of collaborative filtering:

- **User-based Collaborative Filtering**: Recommends items by finding similar users to the target user and recommending items those similar users liked.
- **Item-based Collaborative Filtering**: Recommends items similar to the items the user has already liked.

In this example, we'll focus on **Matrix Factorization**, which is a popular collaborative filtering technique.

Step 1: Install Libraries `pip install numpy pandas`

`scikit-learn surprise`

The `surprise` library is a Python library that implements collaborative filtering and matrix factorization methods.

Step 2: Prepare the Data

We'll use a **movie ratings dataset** as an example. You can use a dataset like **MovieLens**. `from`

```
surprise import Dataset, Reader import

pandas as pd

# Load the MovieLens dataset (example with ratings) url =
"https://raw.githubusercontent.com/sidooms/MovieTweetings/master/latest/ratings.dat"
ratings = pd.read_csv(url, sep="::", header=None, names=["user_id", "item_id",
"rating", "timestamp"])

# Prepare the data for surprise reader = Reader(rating_scale=(1, 5)) data =
Dataset.load_from_df(ratings[['user_id', 'item_id', 'rating']], reader)
```

- **`ratings`** contain user-item interactions, such as movie ratings.
- **`Dataset.load_from_df()`** prepares the data for use in the collaborative filtering model.

Step 3: Apply Collaborative Filtering using Matrix Factorization (SVD) `from`

```
surprise import SVD

from surprise.model_selection import train_test_split from
surprise import accuracy

# Split the dataset into train and test  trainset,
testset = train_test_split(data, test_size=0.2)

# Use Singular Value Decomposition (SVD) svd =
SVD()

# Train the model svd.fit(trainset)

# Test the model  predictions =
svd.test(testset)

# Evaluate the accuracy accuracy.rmse(predictions)
```

- **SVD (Singular Value Decomposition)**: A matrix factorization technique used in collaborative filtering. It decomposes the user-item interaction matrix into latent factors, which represent hidden relationships between users and items.

Step 4: Make Recommendations

To make a recommendation for a specific user, we can predict ratings for all items and suggest the ones with the highest predicted ratings.

```
def recommend(user_id, n=10):     # Get a
list of all item IDs      all_items =
ratings['item_id'].unique()

    # Predict ratings for each item
    predictions = [svd.predict(user_id, item_id) for item_id in all_items]
    # Sort predictions by rating (descending order)      sorted_predictions =
sorted(predictions, key=lambda x: x.est, reverse=True)
    # Get the top n recommended items     top_n
= sorted_predictions[:n]
    return [(pred.iid, pred.est) for pred in top_n]

# Example: Recommend 10 items for user 1 recommendations
= recommend(user_id=1, n=10) print(recommendations)
```

2. Deep Learning Approaches for Recommendation Systems

Deep learning-based recommendation systems aim to use more complex models, such as neural networks, to capture hidden patterns in user-item interactions. One popular architecture is the **Autoencoder** or a **Neural Collaborative Filtering (NCF)** model.

Step 1: Install Required Libraries `pip install`

```
tensorflow numpy pandas scikit-learn
```

Step 2: Data Preparation

You can use the same dataset as in the collaborative filtering example. We will create a user-item matrix where each row represents a user, and each column represents an item. The values will be the ratings.

```
import numpy as np

# Create user-item matrix
user_item_matrix = ratings.pivot(index='user_id', columns='item_id',
values='rating').fillna(0)
```

- **`pivot()`**: This creates a matrix where rows represent users, columns represent items, and values are the ratings.

Step 3: Build the Neural Network Model (Autoencoder)

```
import tensorflow as tf  from
tensorflow.keras import layers, models

# Define the model architecture (Autoencoder) def create_model(input_dim,
encoding_dim):
    # Input layer
    input_layer = layers.Input(shape=(input_dim,))

    # Encoder
    encoded = layers.Dense(encoding_dim, activation='relu')(input_layer)
    # Decoder
    decoded = layers.Dense(input_dim, activation='sigmoid')(encoded)

    # Autoencoder model
    autoencoder = models.Model(input_layer, decoded)

    # Encoder model (to extract the compressed features)     encoder
= models.Model(input_layer, encoded)

    # Compile the model
    autoencoder.compile(optimizer='adam', loss='mean_squared_error')
         return autoencoder,
encoder  # Set parameters
input_dim =
user_item_matrix.shape[1]  #
Number of items encoding_dim =
100  # Compressed
representation
```

```
# Create the autoencoder model
autoencoder, encoder = create_model(input_dim, encoding_dim)
# Train the autoencoder
autoencoder.fit(user_item_matrix.values, user_item_matrix.values, epochs=50,
batch_size=256, shuffle=True, validation_data=(user_item_matrix.values,
user_item_matrix.values))
```

Autoencoder: An unsupervised neural network used for dimensionality reduction. The encoder part compresses the input (user-item matrix) into a lower-dimensional representation, while the decoder reconstructs the original input.

Step 4: Make Recommendations

Once the autoencoder is trained, we can generate recommendations for users by using the encoder to get the compressed representation of the user-item matrix.

```
# Get the compressed user-item matrix from the encoder encoded_matrix
= encoder.predict(user_item_matrix.values)
#  Recommend  items  for  a  user  (use  the  reconstruction  error  as  a  measure)  def
recommend_deep(user_id, n=10):     user_encoded = encoded_matrix[user_id - 1]  # Get
the encoding for the specific user
    predictions = np.dot(user_encoded, encoded_matrix.T)  # Calculate predicted
ratings for each item

    # Sort by predicted ratings (descending)
    recommended_items = np.argsort(predictions)[::-1][:n]
         return
recommended_items

# Example: Recommend 10 items for user 1 recommended_items =
recommend_deep(user_id=1, n=10) print("Recommended items for
user 1:", recommended_items)
```

- **Autoencoder-based Recommendations**: We use the encoder to obtain compressed user-item representations and then calculate the dot product between the compressed user vector and all other item vectors to predict ratings.

Conclusion

Both **Collaborative Filtering** and **Deep Learning Approaches** are widely used for building recommendation systems, and each has its strengths:

- **Collaborative Filtering** is a traditional method based on user-item interactions. It works well for small datasets and has simpler models (e.g., matrix factorization with SVD).
- **Deep Learning Approaches** (e.g., Autoencoders, Neural Collaborative Filtering) are more advanced and can capture complex relationships in the data, making them more suitable for large datasets with more intricate patterns.

M.Sc. IT. SEM 3 **Advance Artificial Intelligence**

PRACTICAL – 5

Aim: Implementing a Computer Vision project, such as Object Detection or Image Segmentation.

Project 1: Object Detection using TensorFlow and OpenCV

Object detection involves detecting instances of objects within an image and classifying them. The object detection model will output the locations (bounding boxes) of these objects as well as their labels (e.g., person, car, etc.).

We will use **TensorFlow's Object Detection API** to implement object detection. This library provides pre-trained models for different object detection tasks.

Step 1: Install Dependencies

First, install the required libraries.

`pip install tensorflow opencv-python opencv-python-headless` Additionally, we

will install the **TensorFlow Object Detection API**.

`pip install tf-slim pip install tensorflowobject-detection-api`

You also need to install the following dependencies to run the Object Detection API: `pip`

`install --upgrade setuptools pip`

`install pycocotools`

Step 2: Load Pre-trained Model

TensorFlow provides pre-trained models, such as **Faster R-CNN**, **SSD**, and **YOLO**, for object detection. For simplicity, we'll use the **SSD MobileNet V2** model, which is fast and performs well.

```
import tensorflow as tf
import numpy as np import
os import cv2
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as vis_util
# Load pre-trained model
PATH_TO_CKPT = 'ssd_mobilenet_v2_coco_2018_03_29/frozen_inference_graph.pb' model =
tf.saved_model.load(PATH_TO_CKPT)

# Load label map (for COCO dataset)  PATH_TO_LABELS
= 'mscoco_label_map.pbtxt'
category_index = label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
use_display_name=True)
```

- **`ssd_mobilenet_v2_coco_2018_03_29/frozen_inference_graph.pb`** is a frozen model file (you can download it from the TensorFlow model zoo).

- **`mscoco_label_map.pbtxt`** is the label map that corresponds to the COCO dataset classes.

Step 3: Prepare the Image

Now, we'll prepare the input image and run it through the model.

```
# Load an image  image_path = 'image.jpg'  # Path to
input image image_np
= cv2.imread(image_path)

# Convert image to RGB (OpenCV loads in BGR by default) image_rgb =
cv2.cvtColor(image_np, cv2.COLOR_BGR2RGB)

# Convert to a tensor
input_tensor = tf.convert_to_tensor(image_rgb) input_tensor =
input_tensor[tf.newaxis,...]  # Add batch dimension
```

Step 4: Perform Object Detection

We now perform object detection on the input image.

```
# Run detection  model_fn =
model.signatures['serving_default'] output_dict
= model_fn(input_tensor)

# The model outputs various results:
# 'num_detections', 'detection_boxes', 'detection_scores', 'detection_classes'
num_detections = int(output_dict['num_detections'][0]) detection_boxes =
output_dict['detection_boxes'][0].numpy() detection_scores =
output_dict['detection_scores'][0].numpy()
detection_classes = output_dict['detection_classes'][0].numpy().astype(np.int32)
```

- **`detection_boxes`**: Bounding boxes for each object detected.
- **`detection_scores`**: Confidence scores for each object detected.
  `detection_classes`: Class IDs of detected objects.

Step 5: Visualize the Results

We can visualize the detected objects using OpenCV and TensorFlow's visualization utilities.

```
# Visualize detected boxes and labels on the image
vis_util.visualize_boxes_and_labels_on_image_array(
image_np,     detection_boxes,     detection_classes,
detection_scores,     category_index,
instance_masks=None,
use_normalized_coordinates=True,     line_thickness=8
)
# Convert image back to BGR for OpenCV display image_bgr
= cv2.cvtColor(image_np, cv2.COLOR_RGB2BGR)

# Display the output image
cv2.imshow('Object Detection', image_bgr)
cv2.waitKey(0) cv2.destroyAllWindows()
```

- **`visualize_boxes_and_labels_on_image_array()`** draws the bounding boxes on the image, along with labels and confidence scores.

Step 6: Saving the Result

You can also save the resulting image with the detected objects.

```
# Save the output image cv2.imwrite('output_image.jpg', image_bgr)
```

Project 2: Image Segmentation using U-Net

Image segmentation is the task of classifying each pixel in an image as belonging to a specific class (e.g., background, object, etc.). In this example, we'll use a **U-Net** architecture, which is widely used for segmentation tasks.

We'll use the **Keras** library to implement U-Net for semantic segmentation.

Step 1: Install Dependencies

If you don't have **Keras** or **TensorFlow** installed: `pip install`

`tensorflow`

Step 2: Build the U-Net Model

Here's a simplified U-Net model built using **Keras**.

```
import tensorflow as tf
from tensorflow.keras import layers, models
# Build the U-Net model def
unet_model(input_size=(256, 256, 3)):
inputs = layers.Input(input_size)

    # Encoder
    conv1 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(inputs)
conv1 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(conv1)     pool1
= layers.MaxPooling2D((2, 2))(conv1)
    # Decoder
    up1 = layers.UpSampling2D((2, 2))(pool1)
concat1 = layers.concatenate([conv1, up1], axis=-1)
    conv2 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(concat1)
    # Output layer
    outputs = layers.Conv2D(1, (1, 1), activation='sigmoid')(conv2)
          model = models.Model(inputs,
outputs)
model.compile(optimizer='adam',
loss='binary_crossentropy',
metrics=['accuracy'])
      return
model

# Create the U-Net model model
= unet_model() model.summary()
```

- **`Conv2D`**: Convolutional layers for feature extraction.
- **`MaxPooling2D`**: Downsampling the feature map.
- **`UpSampling2D`**: Upsampling to reconstruct the segmentation map. **`concatenate`**: Skip connections to preserve spatial information.

Step 3: Train the Model

Now, train the model with a dataset. We can use any segmentation dataset (for example, **Pascal VOC** or **COCO**), or you can create a simple dataset with images and corresponding masks (ground truth segmentation maps). `# Train the model`

```
# X_train: input images, Y_train: segmentation masks model.fit(X_train, Y_train,
batch_size=16, epochs=10)
```

Step 4: Make Predictions

Once the model is trained, you can use it to predict segmentation masks for new images.

```
# Predict segmentation map for a new image segmentation_map
= model.predict(X_test)

# Display the result import matplotlib.pyplot
as plt
plt.imshow(segmentation_map[0, :, :, 0], cmap='gray') plt.show()
```

Step 5: Post-Processing and Visualization

You can further process and visualize the segmentation map by overlaying it on the original image.

```
# Threshold segmentation map to get a binary mask
binary_mask = (segmentation_map[0, :, :, 0] > 0.5).astype(np.uint8)
# Overlay the mask on the original image
segmented_image = cv2.addWeighted(X_test[0], 0.7, binary_mask * 255, 0.3, 0)
# Display the segmented image plt.imshow(segmented_image)
plt.show()
```

Conclusion

- **Object Detection** was implemented using TensorFlow's Object Detection API.
- **Image Segmentation** was implemented using a simple **U-Net** model in Keras.

M.Sc. IT. SEM 3 **Advance Artificial Intelligence**

PRACTICAL – 6

Aim: Training a Generative Adversarial Network (GAN) for Generating Realistic Images

Generative Adversarial Networks (GANs) are a class of machine learning models used to generate synthetic data, including realistic images. A GAN consists of two networks: a **generator** and a **discriminator**. The generator creates images, and the discriminator tries to differentiate between real images (from the dataset) and fake images (from the generator). The generator aims to fool the discriminator, and through this adversarial process, both models improve. **Steps Involved in Training a GAN**

1. **Prepare the dataset**: Load and preprocess the dataset.
2. **Build the GAN architecture**: Create the generator and discriminator networks.
3. **Define the loss functions**: Use binary cross-entropy for both networks.
4. **Train the GAN**: Train both networks in an adversarial manner.
5. **Generate images**: Use the trained generator to produce images.

Step-by-Step Implementation Step 1:

Install Dependencies

First, we need to install the required libraries.

```
pip install tensorflow matplotlib numpy
```

Step 2:

Import Libraries

We will import the necessary libraries for building and training the GAN.

```
import tensorflow as tf from
tensorflow.keras import layers import
numpy as np import matplotlib.pyplot
as plt
```

Step 3: Load and Preprocess the Dataset

We'll use the **MNIST** dataset, which consists of 28x28 grayscale images of handwritten digits (0-9).

```
# Load MNIST dataset
(X_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
# Normalize the images to [-1, 1]
X_train = X_train.astype("float32")
X_train = (X_train - 127.5) / 127.5  # Normalize to the range [-1, 1]
X_train = np.expand_dims(X_train, axis=-1)  # Add a channel dimension
```

- **Normalization**: This is necessary because GANs are more stable when inputs range between [-1, 1].

Step 4: Define the Generator and Discriminator **Generator Model**

The **Generator** takes random noise as input and generates images from it. We will use several **dense** and **conv2d** layers to upsample the noise into an image.

```
def build_generator():
    model = tf.keras.Sequential()
    model.add(layers.InputLayer(input_shape=(100,)))  # Random noise of size 100
    model.add(layers.Dense(7 * 7 * 256, use_bias=False))  # Dense layer
    model.add(layers.BatchNormalization())  # Batch normalization
    model.add(layers.LeakyReLU())  # Leaky ReLU activation

    model.add(layers.Reshape((7, 7, 256)))  # Reshape to a 7x7x256 tensor
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))  # Output layer

    return model

generator = build_generator()
```

- **Dense Layer**: Starts with a fully connected layer that takes a 100-dimensional vector (noise) as input and outputs a flattened 7x7x256 tensor.
- **Conv2DTranspose**: Upsamples the data, converting it into a higher resolution image. **LeakyReLU**: Used for non-linear activation.

Discriminator Model

The **Discriminator** takes an image as input and outputs a single scalar value (0 or 1), indicating whether the image is real (from the dataset) or fake (generated by the generator).

```
def build_discriminator():
    model = tf.keras.Sequential()
    model.add(layers.InputLayer(input_shape=(28, 28, 1)))
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))  # Dropout for regularization

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
```

```
    model.add(layers.Flatten())  # Flatten to 1D vector
    model.add(layers.Dense(1, activation='sigmoid'))  # Output layer: real or fake
return model

discriminator = build_discriminator()
```

- **Conv2D**: Convolution layers are used to extract features from the image.
- **LeakyReLU**: Activation function that allows small negative values.
- **Dropout**: Used for regularization to prevent overfitting.

Step 5: Define the GAN Model

The **GAN** model is a combination of the generator and discriminator. The generator's goal is to produce images that can fool the discriminator into classifying them as real.

```
# GAN model: The generator produces images, and the discriminator classifies them
discriminator.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# The discriminator's weights are frozen during the training of the GAN
discriminator.trainable = False

gan_input = layers.Input(shape=(100,))
x = generator(gan_input) gan_output = discriminator(x)

gan = tf.keras.Model(gan_input, gan_output)  gan.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
```

- **Frozen Discriminator**: During the GAN training, only the generator is trained. The discriminator's weights are kept frozen to prevent it from being updated during this phase.

Step 6: Training the GAN

We now set up the training loop. The GAN training alternates between training the discriminator and training the generator.

```
import time  def
train_gan(epochs,
batch_size):
    # Real labels are 1 and fake labels are 0     real_labels
= np.ones((batch_size, 1))     fake_labels
= np.zeros((batch_size, 1))
for epoch in
range(epochs):         start_time
= time.time()

        # Train the discriminator on real images
        idx = np.random.randint(0, X_train.shape[0], batch_size)
real_images = X_train[idx]
```

```
        d_loss_real = discriminator.train_on_batch(real_images, real_labels)
# Train the discriminator on fake images generated by the generator
noise = np.random.randn(batch_size, 100)         fake_images =
generator.predict(noise)
        d_loss_fake = discriminator.train_on_batch(fake_images, fake_labels)
        # Calculate the total discriminator loss
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)          #
Train the generator through the GAN model         g_loss =
gan.train_on_batch(noise, real_labels)

        # Print progress
        elapsed_time = time.time() - start_time         print(f"{epoch}/{epochs} | D
Loss: {d_loss[0]} | G Loss: {g_loss[0]} | Time: {elapsed_time:.2f}s")

        # Save generated images periodically         if epoch % 1000
== 0:             save_generated_images(epoch)   def
save_generated_images(epoch, examples=16, dim=(4, 4), figsize=(4, 4)):
noise = np.random.randn(examples, 100)     generated_images =
generator.predict(noise)
     plt.figure(figsize=figsize)     for
i in range(examples):         plt.subplot(dim[0],
dim[1], i+1)
        plt.imshow(generated_images[i, :, :, 0], cmap='gray')
plt.axis('off')     plt.tight_layout()
    plt.savefig(f'generated_image_epoch_{epoch}.png')
plt.close()

# Train the GAN
train_gan(epochs=10000, batch_size=64)
```

- **Training Loop**: ○ **Discriminator**: Trains on both real and fake images. ○ **Generator**: Trains via the GAN model to fool the discriminator into classifying fake images as real.
- **Image Saving**: We periodically save generated images to visualize how the model is improving.

Step 7: Visualize the Results

During training, the generator improves its ability to produce realistic images. After training, you can generate new images.

```
# Generate and visualize new images after training noise
= np.random.randn(16, 100)
generated_images = generator.predict(noise)
plt.figure(figsize=(4, 4)) for i in
range(16):     plt.subplot(4, 4, i+1)
    plt.imshow(generated_images[i, :, :, 0], cmap='gray')
plt.axis('off') plt.tight_layout() plt.show()
```

Conclusion

We've successfully implemented and trained a **Deep Convolutional GAN (DCGAN)** using TensorFlow for generating realistic images. The **generator** creates fake images, and the **discriminator** attempts to classify them as real or fake.

.

PRACTICAL – 7

Aim: Applying Reinforcement Learning (RL) Algorithms to Solve Complex DecisionMaking Problems

Reinforcement Learning (RL) is a type of machine learning where an agent learns how to make decisions by interacting with an environment. The goal is to learn an optimal policy that maximizes cumulative rewards over time. In RL, an agent takes actions in an environment and receives feedback in the form of rewards or penalties, allowing it to adjust its actions accordingly.

Problem: Solving a Grid World Problem with Q-Learning

A **Grid World** is a simple environment where an agent moves around a grid to reach a goal. The grid can have obstacles, and the agent must learn to navigate it to maximize rewards (for example, reaching the goal in the fewest steps).

We'll demonstrate the application of **Q-Learning** in a simple grid world setup.

Steps:

1. **Set up the environment**: Define the grid world with rewards, states, and actions.
2. **Define the Q-learning algorithm**: Define how the agent learns the optimal policy.
3. **Train the agent**: Let the agent explore the environment and update its Q-values.
4. **Evaluate the learned policy**: Test the agent after training.

Step-by-Step Implementation

Step 1: Install Required Libraries

You need **NumPy** for array manipulation. Install it if you don't have it. `pip install`

```
numpy
```

Step 2: Set Up the Environment

We will create a grid of size 5x5. The agent starts at the top-left corner (0,0) and must reach the goal at the bottom-right corner (4,4).

```
import numpy as np

# Define the grid world grid_size = 5  # 5x5 grid goal = (4, 4)  # Goal
position  obstacles = [(1, 1), (2, 1), (3, 3)]  # Obstacles that the agent
cannot pass
# Define the rewards for each cell reward_grid = np.zeros((grid_size,
grid_size)) reward_grid[goal] = 1  #
Positive reward at the goal for obs in obstacles:
```

```
reward_grid[obs] = -1  # Negative reward for obstacles  # Define actions: Up, Down,
Left, Right  actions = [(0, -1), (0, 1), (-1, 0), (1, 0)]  # (dy, dx) for up, down,
left, right  # Helper function to check if a position is within the grid and not
an obstacle def is_valid_move(pos):     if 0 <= pos[0] < grid_size and 0 <= pos[1]
< grid_size and pos not in obstacles:          return True
return False
```

Step 3: Q-Learning Algorithm

Q-learning is an off-policy RL algorithm. It learns the value of state-action pairs (Q-values), and the policy is derived by selecting the action with the highest Q-value in each state.

The Q-learning update rule is:

Q(s,a)=Q(s,a)+α[r+γmax⁡a′Q(s′,a′)−Q(s,a)]Q(s, a) = Q(s, a) + \alpha \left[r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a) \right]

Where:

- Q(s,a)Q(s, a) is the Q-value for state ss and action aa,
- α\alpha is the learning rate,
- γ\gamma is the discount factor,
- rr is the reward for taking action aa from state ss,
- s′s' is the new state after taking action aa, a′a' is the next action chosen.

```
# Initialize Q-table with zeros
Q = np.zeros((grid_size, grid_size, len(actions)))  # Q(s, a)
# Hyperparameters alpha = 0.1  #
Learning rate gamma = 0.9  #
Discount factor epsilon = 0.1  # Exploration
rate  episodes = 1000  # Number of training
episodes
max_steps = 100  # Maximum steps per episode
# Training the agent def
train_q_learning():     for
episode in range(episodes):
        state = (0, 0)  # Start at the top-left corner         total_reward
= 0
                for step in range(max_steps):             # Exploration vs.
Exploitation             if np.random.rand() < epsilon:
action_idx = np.random.choice(len(actions))  # Random action  (exploration)
else:                 action_idx = np.argmax(Q[state[0], state[1]])  #
Greedy action
(exploitation)
            # Get the new state after taking the action             action
= actions[action_idx]
            next_state = (state[0] + action[0], state[1] + action[1])
            # Check if the move is valid             if
not is_valid_move(next_state):
next_state = state  # Stay in the same position if
invalid move
```

```
            # Get reward for the new state             reward
= reward_grid[next_state]

            # Q-value update
            best_next_action = np.max(Q[next_state[0], next_state[1]])  # max_a'
Q(s', a')
            Q[state[0], state[1], action_idx] += alpha * (reward + gamma *
best_next_action - Q[state[0], state[1], action_idx])

            # Update state
state = next_state             total_reward
+= reward

            # Check if we reached the goal             if
state == goal:                 break

        # Print progress every 100 episodes         if episode % 100 == 0:
print(f"Episode {episode}/{episodes}, Total Reward: {total_reward}")
train_q_learning()
```

Step 4: Evaluate the Learned Policy

Once training is complete, we can evaluate the agent's learned policy by simulating its actions in the grid.

```
# Function to evaluate the learned policy def
evaluate_policy():     state = (0, 0)  # Start at
the top-left corner
path = [state]  # To track the path taken by the agent
     while state !=
goal:          action_idx = np.argmax(Q[state[0], state[1]])  # Choose the best
action
(greedy)
        action = actions[action_idx]
        next_state = (state[0] + action[0], state[1] + action[1])
        # Check if the move is valid         if not
is_valid_move(next_state):             next_state = state  # Stay in the same
position if invalid move

        state = next_state
path.append(state)

    return path

# Visualize the path taken by the agent path
= evaluate_policy()  print("Path taken
by the agent:", path)

# Create a grid visualization def visualize_path(path):
    grid = np.zeros((grid_size, grid_size), dtype=int)
for (y, x) in path:          grid[y, x]
= 1  # Mark the path      # Print grid
with path     for row in grid:
```

```
print(" ".join(["#" if cell == 1 else
"." for cell in row]))
visualize_path(path)
```

Explanation:

1. **Grid Setup**: The grid is represented by a 5x5 matrix. The agent starts at the top-left (0, 0) and the goal is at (4, 4). Obstacles are placed at certain positions, and the agent cannot pass through them.
2. **Q-Learning Algorithm**:
   - The agent starts at (0, 0) and moves around the grid. ○ The agent uses **epsilon-greedy** policy: It either explores (random action) or exploits (chooses the best-known action).
   - For each action, the Q-value is updated using the **Q-learning update rule**.
   - The agent continues training for a specified number of episodes (1000 in this case).
3. **Evaluation**:
   - After training, we evaluate the agent by letting it follow the learned policy. The agent chooses actions greedily based on the Q-values.
   - The path the agent takes from start to goal is displayed.
4. **Result Visualization**: The grid with the path the agent took is visualized, where `#` represents the path the agent took.

Conclusion:

This simple example demonstrates how **Q-Learning** can be applied to a decision-making problem (navigating a grid) where the agent learns to maximize cumulative rewards (reaching the goal). By exploring and exploiting the environment, the agent updates its Q-values and improves its policy over time.

PRACTICAL – 8

Aim: Utilizing Transfer Learning to Improve Model Performance on Limited Datasets

Transfer learning is a powerful technique in deep learning where knowledge gained from training a model on a large, diverse dataset is applied to a new, typically smaller dataset. It leverages pre-trained models, which have already learned features that can be reused for similar tasks, thereby improving the performance on the new task with relatively less data.

Transfer learning is particularly useful when the new dataset is limited or lacks sufficient labeled data for training a model from scratch.

Transfer Learning Using a Pre-Trained Model

Step 1: Install Dependencies

If you haven't installed TensorFlow, use the following command: `pip install`

```
tensorflow
```

Step 2: Load a Pre-Trained Model

We will use the **ResNet50** model, which is a widely used architecture pre-trained on the **ImageNet** dataset. We'll load it without the top classification layers so we can adapt it to our own task.

```
import tensorflow as tf
from tensorflow.keras import layers, models from
tensorflow.keras.applications import ResNet50
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# Load the pre-trained ResNet50 model, without the top layer (classification layer)
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224,
3))
# Freeze the base model layers to prevent them from being updated during training
base_model.trainable = False
```

Step 3: Add Custom Layers

After the pre-trained model, we add custom layers that are specific to the new task. These layers will learn from the new dataset.

```
# Create a custom model by adding layers on top of the base ResNet50 model model =
models.Sequential()

# Add the pre-trained ResNet50 model model.add(base_model)
```

```
# Add custom layers (GlobalAveragePooling2D, Dense layer, and output layer)
model.add(layers.GlobalAveragePooling2D()) model.add(layers.Dense(1024,
activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))  # For binary classification (adjust
for multi-class)

# Compile the model  model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
```

Step 4: Prepare the Dataset

Since the ResNet50 model expects input images of size **224x224x3**, we'll preprocess the images accordingly. You can use an existing dataset or a custom dataset of images. For the sake of illustration, let's assume you're using a binary classification problem with limited data.

To augment the data (using techniques like rotation, flipping, and scaling), we use **ImageDataGenerator**.

```
# Use ImageDataGenerator for real-time data augmentation train_datagen
= ImageDataGenerator(     rescale=1./255,  # Normalize image
pixel values to [0, 1]
rotation_range=40,     width_shift_range=0.2,
height_shift_range=0.2,     shear_range=0.2,     zoom_range=0.2,
horizontal_flip=True,     fill_mode='nearest'
)

# Define validation data generator  validation_datagen
= ImageDataGenerator(rescale=1./255)

# Prepare the data from directories (use appropriate paths for your dataset)
train_generator = train_datagen.flow_from_directory(
    'path_to_train_data',  # Specify the path to the training data
target_size=(224, 224),     batch_size=32,
    class_mode='binary'  # Change this to 'categorical' for multi-class
classification
)  validation_generator =
validation_datagen.flow_from_directory(
    'path_to_validation_data',  # Specify the path to the validation data
target_size=(224, 224),     batch_size=32,
    class_mode='binary'  # Change this to 'categorical' for multi-class
classification
)
```

Step 5: Fine-Tuning the Model

After training the top layers for a few epochs, you can fine-tune the pre-trained layers by unfreezing some of the layers in the base model.

```
# Unfreeze some layers of the base model for fine-tuning base_model.trainable =
True
# Fine-tune from a certain layer onwards (e.g., unfreeze the last 10 layers) for
layer in base_model.layers[:-10]:
    layer.trainable = False

# Recompile the model after unfreezing the layers
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
loss='binary_crossentropy', metrics=['accuracy'])
# Fine-tune the model history
= model.fit(     train_generator,
    epochs=10,  # Number of epochs to fine-tune the model
validation_data=validation_generator )
```

Step 6: Evaluate the Model

After training and fine-tuning, evaluate the model on the validation data to see how well it performs.

```
# Evaluate the model  test_loss, test_acc =
model.evaluate(validation_generator) print(f'Test Accuracy:
{test_acc}')
```

Step 7: Save the Model

Once the model is trained, you can save it for later use or deployment.

```
# Save the model  model.save('transfer_learning_model.h5')
```

Benefits of Transfer Learning

- **Improved Performance on Small Datasets**: Transfer learning allows us to achieve better results on limited data by leveraging pre-trained models that already have learned rich features.
- **Faster Training**: Since the base model has already been trained, we can freeze most of its layers, reducing the training time and computational cost.
- **Lower Risk of Overfitting**: Fine-tuning a pre-trained model helps prevent overfitting, as the model starts with weights that already capture general features from a large dataset (e.g., ImageNet).

Conclusion

Using transfer learning, we can dramatically improve model performance, especially on tasks with limited datasets, by leveraging the knowledge from large-scale pre-trained models. Fine-tuning these models for specific tasks helps in adapting them to new challenges, such as classifying images in a small dataset.

M.Sc. IT. SEM 3 **Advance Artificial Intelligence**

PRACTICAL – 9

Aim: Building a Deep Learning Model for Time Series Forecasting or Anomaly Detection

Time series forecasting and anomaly detection are crucial tasks in various fields like finance, healthcare, and industry. Deep learning models, such as **Recurrent Neural Networks (RNNs)**, **Long Short-Term Memory (LSTM)** networks, and **Gated Recurrent Units (GRUs)**, are particularly suited for time series tasks because they are designed to handle sequential data.

1. **Build a deep learning model for time series forecasting** using **LSTM** (a type of RNN).
2. **Build a deep learning model for anomaly detection** in time series using an **autoencoder**.

Both models will use **TensorFlow** and **Keras**.

Part 1: Time Series Forecasting with LSTM

In time series forecasting, the goal is to predict future values of a time series based on past data.

Step 1: Install Dependencies

Install the necessary Python libraries if you haven't already:

```
pip install tensorflow numpy pandas matplotlib scikit-learn
```

Step 2: Load and Preprocess Data

We'll use the **Airline Passengers Dataset** as an example for time series forecasting. It contains monthly total passenger counts from 1949 to 1960. We will predict future passenger counts based on past data.

```
import numpy as np import pandas
as pd import matplotlib.pyplot
as plt
from sklearn.preprocessing import MinMaxScaler
# Load the dataset (you can replace this with any other time series dataset) url
=
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/airlinepassengers.c
sv"
data = pd.read_csv(url, usecols=[1], engine='python', header=0)
# Visualize the data plt.plot(data) plt.title('Monthly
Airline Passengers')
plt.xlabel('Month') plt.ylabel('Passengers')
plt.show()

# Normalize the data (scaling to [0, 1] range for LSTM) scaler
= MinMaxScaler(feature_range=(0, 1)) data_scaled
= scaler.fit_transform(data)

# Function to create the dataset in X (input) and y (output) for LSTM
```

```
def create_dataset(data, time_step=1):     X,
y = [], []     for i in range(len(data)-
time_step-1):
X.append(data[i:(i+time_step), 0])
        y.append(data[i + time_step, 0])
return np.array(X), np.array(y)

# Prepare the data for LSTM
time_step = 12  # Use 12 previous months to predict the next month X,
y = create_dataset(data_scaled, time_step)

# Reshape X for LSTM input: [samples, time steps, features]
X = X.reshape(X.shape[0], X.shape[1], 1)
```

Step 3: Build the LSTM Model

Now we define the **LSTM model** architecture:

```
from tensorflow.keras.models import Sequential  from
tensorflow.keras.layers import LSTM, Dense, Dropout

# Build the LSTM model model
= Sequential()  model.add(LSTM(units=50, return_sequences=False,
input_shape=(X.shape[1], 1)))
model.add(Dropout(0.2))  # Dropout for regularization
model.add(Dense(units=1))  # Output layer (single value for forecasting)
# Compile the model model.compile(optimizer='adam', loss='mean_squared_error')
```

Step 4: Train the Model

We will train the model on the time series data:

```
# Train the model model.fit(X, y, epochs=20,
batch_size=32)
```

Step 5: Make Predictions

Now that the model is trained, we can use it to make predictions:

```
# Predict the next 12 months
test_input = data_scaled[-time_step:]  # Last 12 months of data for forecasting
test_input = test_input.reshape(1, -1, 1)  # Reshape for LSTM input
# Predict
predicted = model.predict(test_input)  predicted =
scaler.inverse_transform(predicted)  # Inverse scaling
print(f'Predicted passengers for the next month:
{predicted[0][0]}')
```

Step 6: Visualize the Predictions

```
# Plot the results
train_data = data[:len(data) - 12] plt.plot(train_data,
```

```
label='Training Data')
plt.plot(range(len(train_data), len(train_data) + 12), predicted, label='Forecast',
color='red')
plt.legend()
plt.title('Time Series Forecasting')
plt.xlabel('Month') plt.ylabel('Passengers')
plt.show()
```

Anomaly Detection with Autoencoders

Anomaly detection in time series is the task of identifying unusual patterns or outliers. An **autoencoder** is an unsupervised deep learning model used for anomaly detection. The autoencoder learns to compress (encode) the input into a latent representation and then reconstruct (decode) it. If the reconstruction error is high, the data point is likely an anomaly.

Step 1: Build the Autoencoder Model

We will build an autoencoder for anomaly detection using time series data. `from`

```
tensorflow.keras.layers import Input, Dense from

tensorflow.keras.models import Model

# Define the autoencoder model input_dim
= X.shape[1]  encoding_dim = 14  # Number of dimensions for the encoded
representation

input_layer = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_layer)
decoded = Dense(input_dim, activation='sigmoid')(encoded)
# Create the autoencoder model  autoencoder =
Model(inputs=input_layer, outputs=decoded)

# Compile the model  autoencoder.compile(optimizer='adam',
loss='mean_squared_error')

# Train the autoencoder on the normal (non-anomalous) data autoencoder.fit(X, X,
epochs=50, batch_size=32)
```

Step 2: Detect Anomalies

We can now detect anomalies by checking the reconstruction error. If the error exceeds a certain threshold, the data point is considered anomalous.

```
# Make predictions using the autoencoder reconstructed
= autoencoder.predict(X)

# Calculate reconstruction error (Mean Squared Error)  reconstruction_error
= np.mean(np.square(X - reconstructed), axis=1)
# Set a threshold for anomaly detection (this can be tuned) threshold =
np.percentile(reconstruction_error, 95)
```

```
# Identify anomalies (where reconstruction error is higher than threshold) anomalies
= reconstruction_error > threshold
```

```
# Visualize anomalies plt.plot(data)
plt.scatter(np.where(anomalies)[0], data[anomalies], color='red', label='Anomalies')
plt.title('Anomaly Detection in Time Series') plt.xlabel('Time') plt.ylabel('Value')
plt.legend() plt.show()
```

Step 3: Evaluate the Results

To evaluate the anomaly detection, you could:

- Use labeled datasets with known anomalies.
- Analyze the identified anomalies and compare them with known outliers.

Conclusion

- **LSTM for Forecasting**: This model is suitable for forecasting future values in time series data. It captures temporal dependencies and is effective when historical data patterns are crucial to predict future outcomes.
- **Autoencoders for Anomaly Detection**: Autoencoders are great for detecting anomalies in time series data because they learn the typical patterns and can highlight unusual or out-of-norm events by examining reconstruction errors.

M.Sc. IT. SEM 3 **Advance Artificial Intelligence**

PRACTICAL – 10

Aim: Implementing a Machine Learning Pipeline for Automated Feature Engineering and Model Selection

A **Machine Learning Pipeline** automates various steps in the process of building and deploying machine learning models. The pipeline typically includes:

1. Data preprocessing and feature engineering
2. Model selection and evaluation
3. Hyperparameter tuning
4. Model training and deployment

Automated Machine Learning Pipeline

We'll demonstrate the pipeline using **scikit-learn** and the **TPOT** library for automated machine learning. TPOT helps automatically select the best features, models, and hyperparameters for your task.

Step 1: Install Dependencies

Install the necessary Python libraries:

```
pip install scikit-learn tpot optuna pandas numpy matplotlib
```

Step 2: Load and Preprocess Data

For the sake of illustration, we'll use the **Iris dataset** from `scikit-learn` (you can replace this with your own dataset):

```
import pandas as pd from
sklearn import datasets
from sklearn.model_selection import train_test_split from
sklearn.preprocessing import StandardScaler

# Load the Iris dataset iris
= datasets.load_iris() X =
iris.data y = iris.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Feature scaling (important for algorithms like SVM and neural networks) scaler =
StandardScaler()
X_train_scaled = scaler.fit_transform(X_train) X_test_scaled =
scaler.transform(X_test)
```

Step 3: Automated Feature Engineering (Optional)

While basic preprocessing like scaling is covered, more advanced feature engineering can be done using libraries like **Feature-engine** or **TPOT**. For this basic example, we'll focus on preprocessing using `scikit-learn`.
If you had more complex data (with categorical variables, missing values, etc.), automated feature engineering might involve:

- Encoding categorical variables (e.g., One-Hot Encoding, Label Encoding)
- Imputation of missing values
- Feature extraction techniques (e.g., PCA, polynomial features, etc.)

Step 4: Use TPOT for Automated Model Selection and Hyperparameter Tuning

TPOT uses genetic algorithms to automatically search for the best model and preprocessing pipeline. Here, we'll use TPOT to automate the selection of machine learning models and hyperparameters.

```
from tpot import TPOTClassifier

# Initialize the TPOTClassifier
tpot = TPOTClassifier( generations=5, population_size=20, random_state=42, cv=5,
verbosity=2)
# Train the model (TPOT automatically handles preprocessing and model selection)
tpot.fit(X_train_scaled, y_train)

# Evaluate the model
accuracy = tpot.score(X_test_scaled, y_test) print(f"Test
accuracy: {accuracy:.4f}")

# Export the best pipeline tpot.export('best_model_pipeline.py')
```

Step 5: Automated Hyperparameter Tuning with Optuna (Optional)

In addition to using TPOT, we can also use **Optuna** for hyperparameter optimization. Optuna allows you to define a search space and automatically search for the best hyperparameters using techniques like **Tree-structured Parzen Estimator (TPE)**.

Here is an example using **Optuna** to optimize hyperparameters for a **Random Forest Classifier**. `import optuna`

```
from sklearn.ensemble import RandomForestClassifier from
sklearn.metrics import accuracy_score

# Define the objective function for optimization def
objective(trial):     # Hyperparameter space
    n_estimators = trial.suggest_int('n_estimators', 50, 200)
max_depth = trial.suggest_int('max_depth', 5, 20)
    min_samples_split = trial.suggest_int('min_samples_split', 2, 10)
```

```
    # Train the model with the selected hyperparameters
    model = RandomForestClassifier(n_estimators=n_estimators, max_depth=max_depth,
min_samples_split=min_samples_split, random_state=42)
model.fit(X_train_scaled, y_train)
    # Predict and calculate accuracy     y_pred
= model.predict(X_test_scaled)     accuracy =
accuracy_score(y_test, y_pred)     return
accuracy

# Create an Optuna study and optimize study =
optuna.create_study(direction="maximize") study.optimize(objective,
n_trials=20)

# Print the best hyperparameters found print("Best
hyperparameters:", study.best_params)

# Train the model with the best hyperparameters best_params
= study.best_params best_model =
RandomForestClassifier(**best_params, random_state=42)
best_model.fit(X_train_scaled, y_train)

# Evaluate the model best_accuracy =
best_model.score(X_test_scaled, y_test) print(f"Best Model Test
Accuracy: {best_accuracy:.4f}")
```

Step 6: Evaluate the Best Model

After training and tuning, you can evaluate the model's performance on the test set. Both **TPOT** and **Optuna** return models with optimized parameters, which you can then use to predict on new data.

```
# Evaluate the best model from TPOT or Optuna y_pred =
best_model.predict(X_test_scaled)   accuracy    = accuracy_score(y_test,
y_pred) print(f"Test Accuracy of
Final Model: {accuracy:.4f}")
```

Step 7: Model Deployment (Optional)

Once you have the best model, you can save it for future use or deployment using `joblib` or `pickle`.

```
import joblib

# Save the trained model joblib.dump(best_model,
'best_model.pkl')

# Save the scaler for future use joblib.dump(scaler, 'scaler.pkl')
```

This allows you to reload the model and use it for predictions on new, unseen data.

INDEX

Sr. No.	Practical Aim	Signature
1.	**Data Pre-processing and Exploration** **a.** Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers. **b.** Load a dataset, calculate descriptive summary statistics, create visualizations using different graphs, and identify potential features and target variables Explore Univariate and Bivariate graphs (Matplotlib) and Seaborn for visualization. **c.** Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization.	
2.	**Testing Hypothesis** **a.** Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a. CSV file and generate the final specific hypothesis. (Create your dataset)	

3.	**Linear Models** **a. Simple Linear Regression** Fit a linear regression model on a dataset. Interpret coefficients, make predictions, and evaluate performance using metrics like R-squared and MSE. **b. Multiple Linear Regression** Extend linear regression to multiple features. Handle feature selection and potential multicollinearity. **c. Regularized Linear Models** (Ridge, Lasso, Elastic Net) Implement regression variants like LASSO and Ridge on any generated dataset		
4.	**Discriminative Models** **a. Logistic Regression** Perform binary classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve**.** **b.** Implement and demonstrate k-nearest Neighbor algorithm. Read the training data from a .CSV file and build the model to classify a test sample. Print both correct and wrong predictions.		

	c. Build a decision tree classifier or regressor. Control hyper parameters like tree depth to avoid overfitting. Visualize the tree. **d.** Implement a Support Vector Machine for any relevant dataset. **e.** Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree. **f.** Implement a gradient boosting machine (e.g., XGBoost). Tune hyper **g.** parameters and explore feature importance.	
5.	**Generative Models** **a.** Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample. **b.** Implement Hidden Markov Models using hmmlearn	
6.	**Probabilistic Models** **a.** Implement Bayesian Linear Regression to explore prior and posterior distribution. **b.** Implement Gaussian Mixture Models for density estimation and unsupervised clustering	
7.	**Model Evaluation and Hyper parameter Tuning** **a.** Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation. **b.** Systematically explore combinations of hyper parameters to optimize model performance.(use grid and randomized search)	
8.	**Bayesian Learning** **a.** Implement Bayesian Learning using inferences	

Practical 1: Data Pre-processing and Exploration

1a. Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers.

Code :

1. **Import Libraries**

 # Import necessary libraries

```
import pandas as pd  import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

2. **Load the Dataset**

 # Load the Titanic dataset from a URL

```
url="https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv" data = pd.read_csv(url)
```

 # Display the first few rows

```
print(data.head())
```

3. **Handle Missing Values**

 # Check for missing values

```
print("Missing values in each column:")
print(data.isnull().sum())
```

 # Fill missing values in 'Age' with the mean

```
data['Age'].fillna(data['Age'].mean(), inplace=True)
```

 # Fill missing values in 'Embarked' with the most common value

```
data['Embarked'].fillna(data['Embarked'].mode()[0], inplace=True)
```

 # Drop rows where 'Cabin' is missing (too many NaNs)

```
data.drop(columns=['Cabin'], inplace=True)
```

 # Verify missing values are handled

```
print("\nAfter handling missing values:")
print(data.isnull().sum())
```

4. Fix Inconsistent Formatting

```
# Fix inconsistent formatting in the 'Sex' column
data['Sex'] = data['Sex'].str.lower().str.strip()
# Verify unique values
print("\nUnique values in 'Sex' column after formatting:")
print(data['Sex'].unique())
```

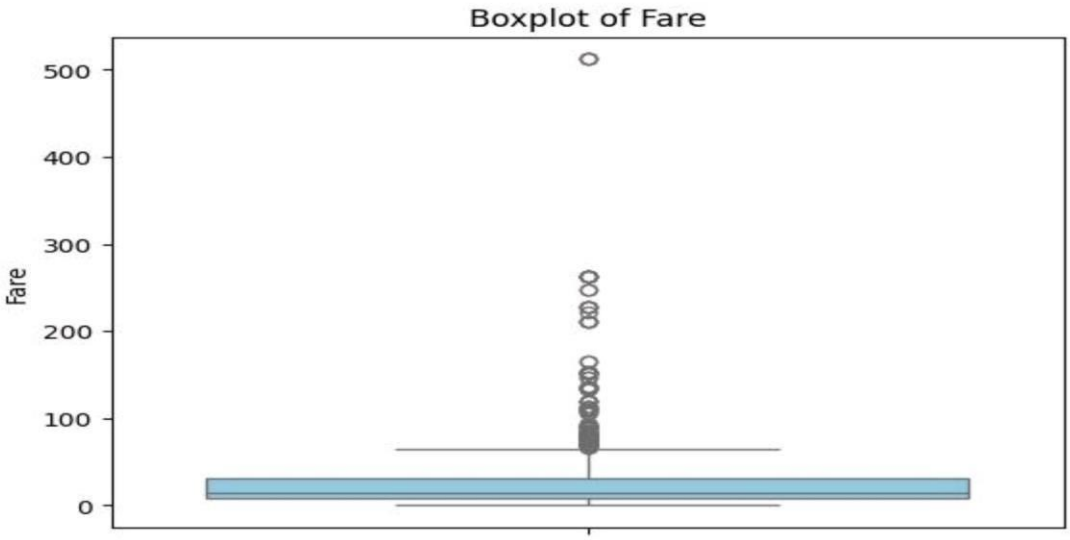
5. Detect and Handle Outliers `# Boxplot for the 'Fare' column sns.boxplot(data['Fare'], color='skyblue') plt.title('Boxplot of Fare') plt.show()`

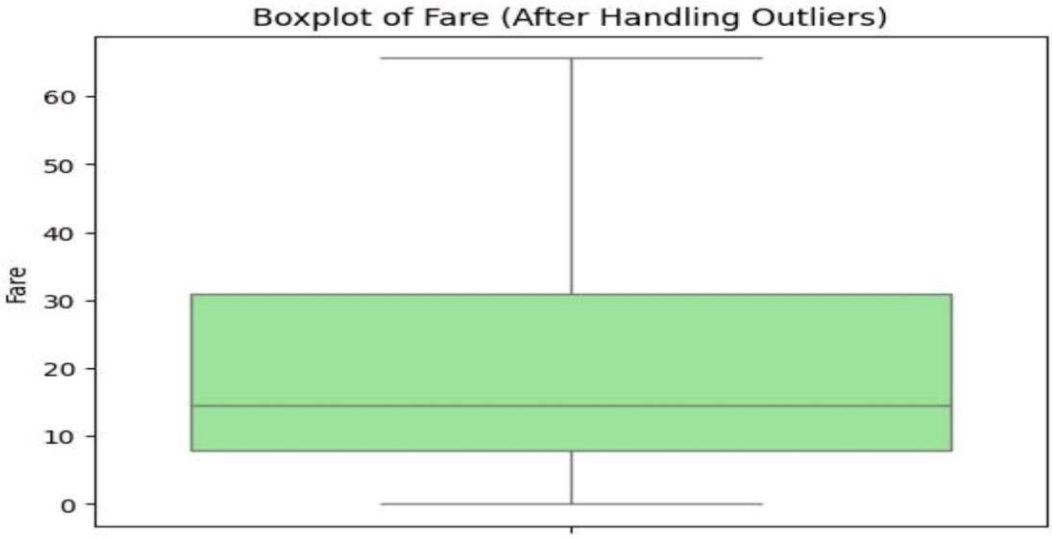
```
# Detect outliers using the IQR method
Q1 = data['Fare'].quantile(0.25)
Q3 = data['Fare'].quantile(0.75)
IQR = Q3 - Q1  lower_bound =
Q1 - 1.5 * IQR  upper_bound =
Q3 + 1.5 * IQR
# Capping outliers
data['Fare'] = np.where(data['Fare'] > upper_bound, upper_bound,   np.where(data['Fare'] <
lower_bound, lower_bound, data['Fare']))
# Verify with an updated boxplot
sns.boxplot(data['Fare'], color='lightgreen')
plt.title('Boxplot of Fare (After Handling Outliers)')
plt.show()
```

6. Save the Cleaned Dataset `# Save the cleaned dataset`

```
data.to_csv('cleaned_titanic.csv', index=False)
print("\nCleaned dataset saved as 'cleaned_titanic.csv'") .
```

Output :


Boxplot of Fare
Fare
500
400
300
200
100
0


Boxplot of Fare (After Handling Outliers)
Fare
60
50
40
30
20
10
0

1b. Load a dataset, calculate descriptive summary statistics, create visualizations using different graphs, and identify potential features and target variables Note:
Explore Univariate and Bivariate graphs (Matplotlib) and Seaborn for visualization

Code :

1. Import Necessary Libraries # Import required libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Load the Dataset

 # Load the dataset from the URL

```
url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv"
data = pd.read_csv(url)
```

 # Display the first few rows

```
print("First 5 rows of the dataset:")
print(data.head())
```

3. Calculate Descriptive Summary Statistics # Dataset information

```
print("\nDataset Info:")
print(data.info())
```

 # Summary statistics for numerical columns

```
print("\nDescriptive Statistics for Numerical Columns:")
print(data.describe())
```

 # Check unique values for categorical columns

```
print("\nUnique values in 'species' column:")
print(data['species'].value_counts())
```

4. Univariate Analysis

 # Histograms for numerical columns

```
data.hist(figsize=(10,8), color='skyblue', edgecolor='black')

plt.suptitle("Histograms of Numerical Features")

plt.show()

# Bar plot for 'species' column
sns.countplot(x='species', data=data, palette='pastel')
plt.title("Count of Each Species")  plt.show()
```

5. Bivariate Analysis

```
# Scatter plot for two features

plt.figure(figsize=(8, 6))

plt.scatter(data['sepal_length'], data['sepal_width'], alpha=0.7, c='blue')

plt.title("Sepal Length vs Sepal Width")

plt.xlabel("Sepal Length")

plt.ylabel("Sepal Width")

plt.show()

# Pairplot to visualize relationships between features
sns.pairplot(data, hue='species', palette='husl', diag_kind='kde')
plt.suptitle("Pairplot of Features by Species", y=1.02)

plt.show()

# Boxplot for petal_length across species sns.boxplot(x='species',
y='petal_length', data=data, palette='Set3')

plt.title("Boxplot of Petal Length by Species")

plt.show()
```

6. Identify Potential Features and Target Variables

```
# Separate features and target

features = data.drop(columns=['species'])

# Drop the target

column  target = data['species']

# Target variable

print("\nFeatures:")

print(features.head())

print("\nTarget:")

print(target.head())
```

 # Visualize target distribution

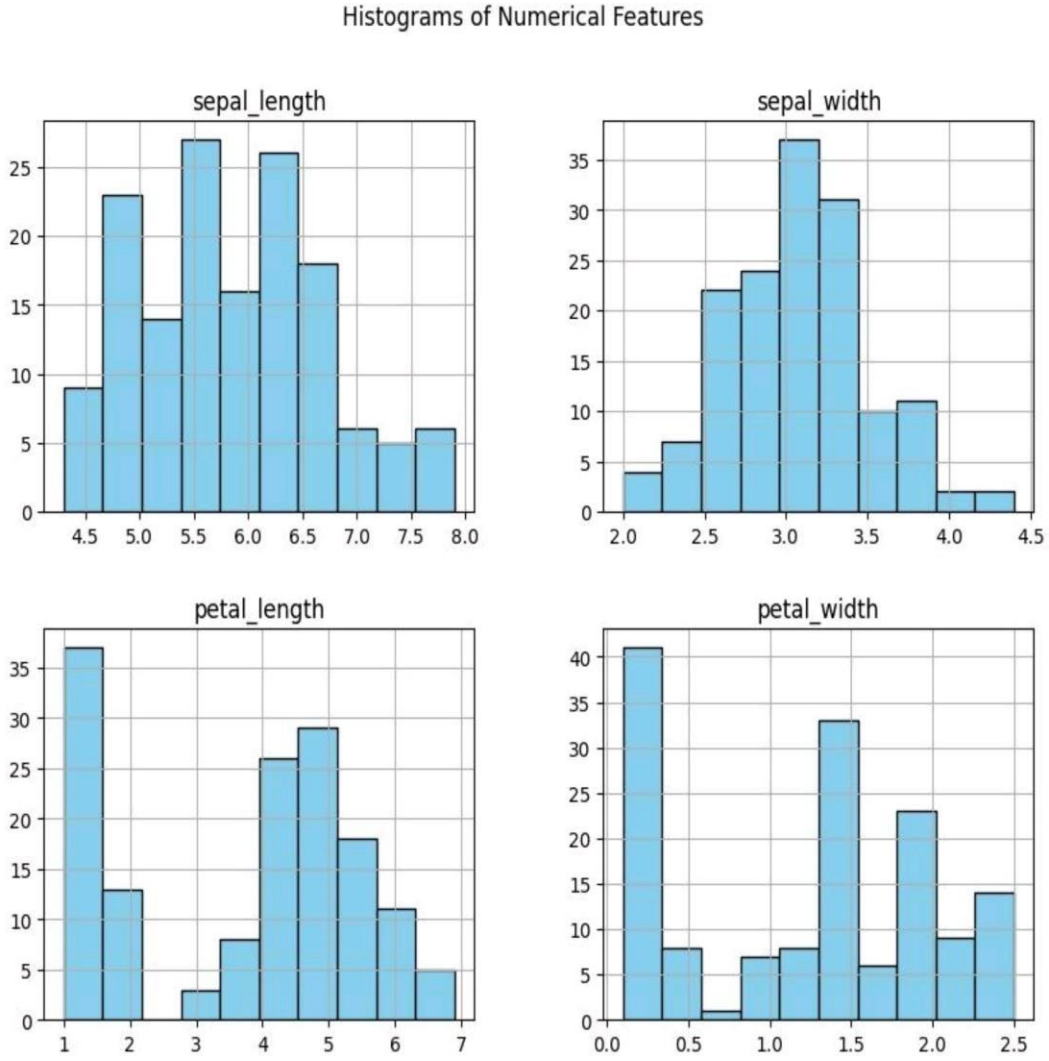
```
sns.countplot(x=target, palette='viridis')
plt.title("Target Variable Distribution")

plt.show()
```

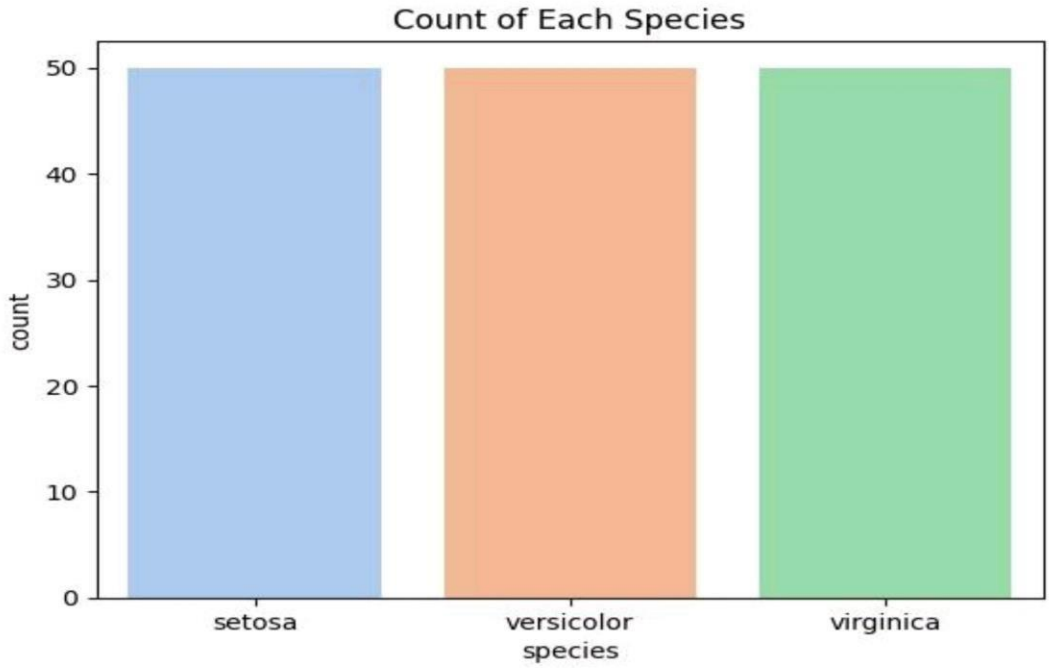
7. Save the Cleaned and Processed Dataset

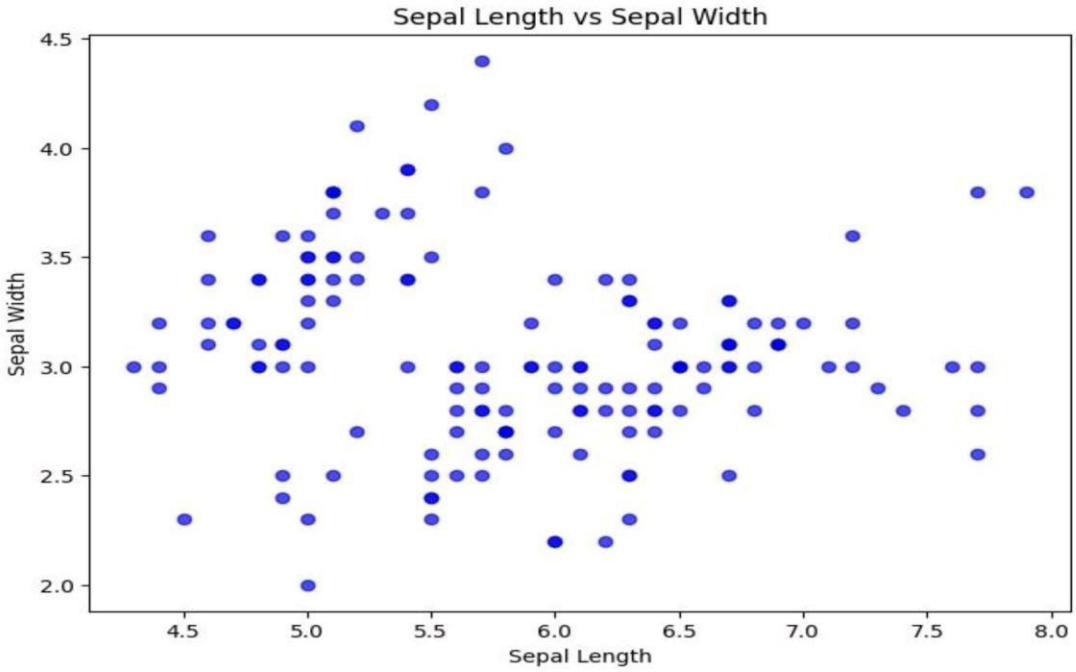
 # Save the dataset

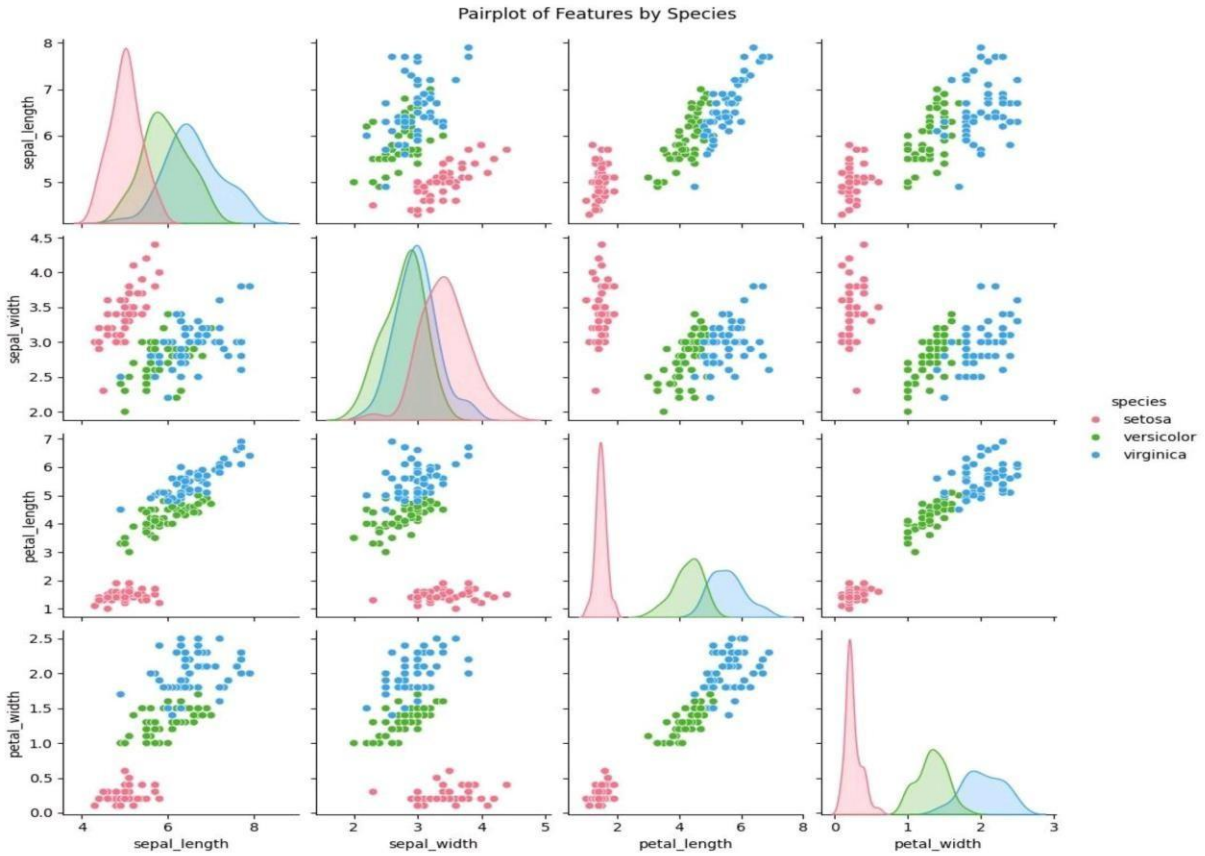
```
data.to_csv('processed_iris.csv', index=False)

print("\nProcessed dataset saved as 'processed_iris.csv'")
```

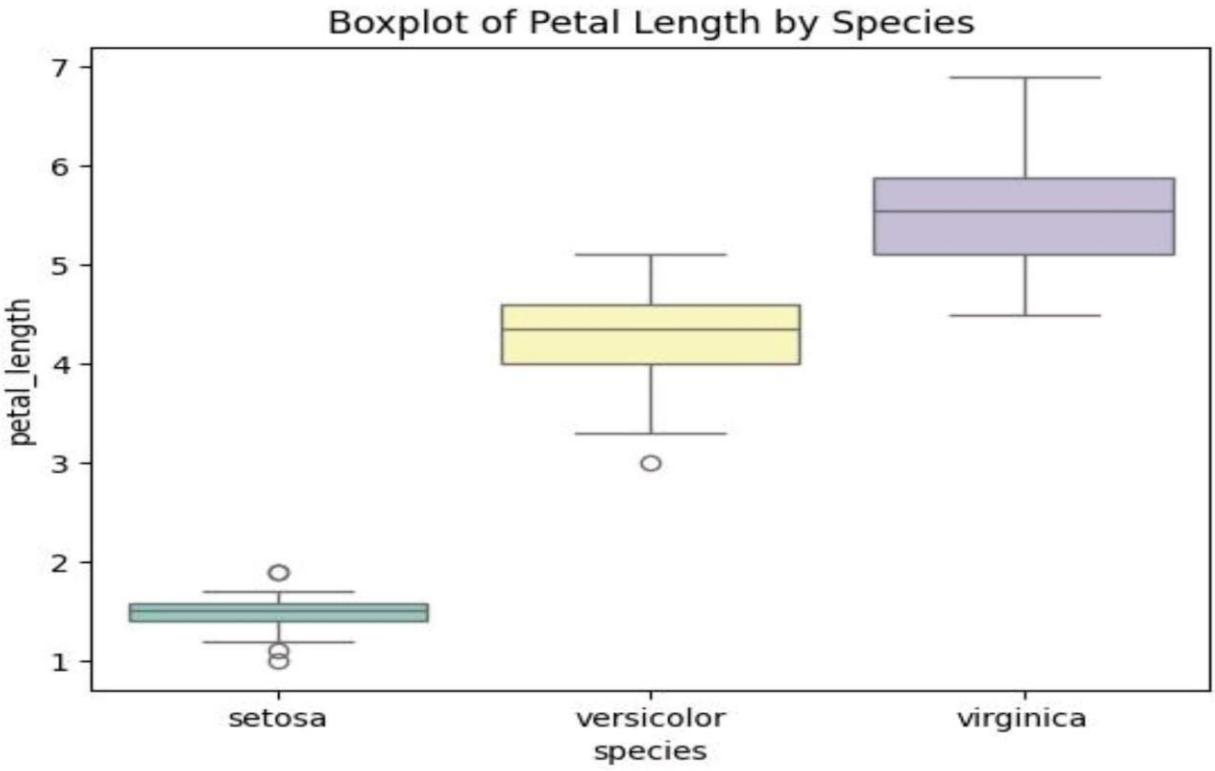
Output :




Count of Each Species
count
50
40
30
20
10
0
setosa
versicolor
virginica
species


Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
Sepal Length


Pairplot of Features by Species
sepal_length
sepal_width
petal_length
petal_width
species
setosa
versicolor
virginica


Boxplot of Petal Length by Species
petal_length
setosa
versicolor
virginica
species

1c. Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization.

Code :

1. Import Necessary Libraries # Import required libraries

```
import pandas as pd
import numpy as np  from sklearn.preprocessing
import LabelEncoder, MinMaxScaler, StandardScaler, Binarizer
```

2. Create or Load a Dataset # Create a sample dataset

```
data = pd.DataFrame({
 'Category': ['A', 'B', 'C', 'A', 'B', 'C'],
# Categorical variable
 'Age': [23, 45, 31, 22, 35, 30],
# Numerical variable
 'Income': [50000, 60000, 70000, 80000, 90000, 100000],
# Numerical variable 'Has_Car':
['Yes', 'No', 'Yes', 'No', 'Yes', 'No']
# Binary categorical variable })

# Display the dataset
print("Sample Dataset:")
print(data)
```

3. Apply Pre-Processing Routines

```
# Label Encoding for 'Category' column
label_encoder = LabelEncoder()
data['Category_Encoded'] =
label_encoder.fit_transform(data['Category'])
# Label Encoding for binary column 'Has_Car'
```

```
data['Has_Car_Encoded'] =
label_encoder.fit_transform(data['Has_Car'])  print("\nAfter Label
Encoding:")

print(data)

# Min-Max Scaling for 'Income'

min_max_scaler = MinMaxScaler()

data['Income_MinMax'] = min_max_scaler.fit_transform(data[['Income']])

# Standard Scaling for 'Age'

standard_scaler = StandardScaler()

data['Age_Standardized'] =

standard_scaler.fit_transform(data[['Age']])  print("\nAfter Scaling:")

print(data)

# Binarization for 'Income' with a threshold of 75,000

binarizer = Binarizer(threshold=75000)

data['Income_Binary'] =
binarizer.fit_transform(data[['Income']])  print("\nAfter
Binarization:")

print(data)
```

4. Save the Processed Dataset

```
# Save the processed dataset
data.to_csv('processed_data.csv', index=False)
print("\nProcessed dataset saved as
'processed_data.csv'")
```

Output :

```
Sample Dataset:
  Category  Age  Income Has_Car
0        A   23   50000     Yes
1        B   45   60000      No
2        C   31   70000     Yes
3        A   22   80000      No
4        B   35   90000     Yes
5        C   30  100000      No
```

```
After Label Encoding:
  Category  Age  Income Has_Car  Category_Encoded  Has_Car_Encoded
0        A   23   50000     Yes                 0                1
1        B   45   60000      No                 1                0
2        C   31   70000     Yes                 2                1
3        A   22   80000      No                 0                0
4        B   35   90000     Yes                 1                1
5        C   30  100000      No                 2                0
```

```
After Scaling:
  Category  Age  Income Has_Car  Category_Encoded  Has_Car_Encoded  \
0        A   23   50000     Yes                 0                1
1        B   45   60000      No                 1                0
2        C   31   70000     Yes                 2                1
3        A   22   80000      No                 0                0
4        B   35   90000     Yes                 1                1
5        C   30  100000      No                 2                0

   Income_MinMax  Age_Standardized
0            0.0         -1.035676
1            0.2          1.812434
2            0.4          0.000000
3            0.6         -1.165136
4            0.8          0.517838
5            1.0         -0.129460
```

```
After Binarization:
  Category  Age  Income Has_Car  Category_Encoded  Has_Car_Encoded  \
0        A   23   50000     Yes                 0                1
1        B   45   60000      No                 1                0
2        C   31   70000     Yes                 2                1
3        A   22   80000      No                 0                0
4        B   35   90000     Yes                 1                1
5        C   30  100000      No                 2                0

   Income_MinMax  Age_Standardized  Income_Binary
0            0.0         -1.035676              0
1            0.2          1.812434              0
2            0.4          0.000000              0
3            0.6         -1.165136              1
4            0.8          0.517838              1
5            1.0         -0.129460              1
```

Practical 2 : Testing Hypothesis

Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a. CSV file and generate the final specific hypothesis. (Create your dataset)

CODE :

```
import pandas as pd

# Step 1: Create the Dataset and Load It

data = {'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rainy', 'Rainy', 'Rainy', 'Overcast', 'Sunny',
'Sunny', 'Rainy'],

'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild'],

'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal',
'Normal'],

'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Weak', 'Weak'],

'PlayTennis': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes']

}


# Load dataset into a pandas DataFrame

df = pd.DataFrame(data)

        # Step 2: Implementing the FIND-S Algorithm

        def find_s_algorithm(data):

            # Get the positive examples (PlayTennis = 'Yes')

            positive_examples = data[data['PlayTennis'] == 'Yes']

            # Initialize hypothesis with the first positive example (most specific)

            hypothesis = positive_examples.iloc[0].drop('PlayTennis')

            # Loop through the rest of the positive examples and generalize the
        hypothesis

            for index, row in positive_examples.iterrows():

                for feature in hypothesis.index:

                    if hypothesis[feature] != row[feature]:

                        hypothesis[feature] = '?'
```

return hypothesis

Step 3: Apply FIND-S to the dataset

hypothesis = find_s_algorithm(df)

 # Display the final specific hypothesis

print("The most specific hypothesis is:")

print(hypothesis)

Output :

```
Dataset:
      Sky Temperature Humidity    Wind Water Forecast Condition
0  Sunny        Warm   Normal  Strong  Warm     Same       Yes
1  Sunny        Cold     High  Strong  Warm     Same        No
2  Rainy        Warm     High    Weak  Cool   Change        No
3  Sunny        Warm   Normal  Strong  Warm     Same       Yes
4  Rainy        Cold   Normal    Weak  Cool   Change        No
```

```
Loaded Dataset:
      Sky Temperature Humidity    Wind Water Forecast Condition
0  Sunny        Warm   Normal  Strong  Warm     Same       Yes
1  Sunny        Cold     High  Strong  Warm     Same        No
2  Rainy        Warm     High    Weak  Cool   Change        No
3  Sunny        Warm   Normal  Strong  Warm     Same       Yes
4  Rainy        Cold   Normal    Weak  Cool   Change        No
```

```
Final Specific Hypothesis:
['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same']
```

Practical 3 : Linear Models

3a. Simple Linear Regression

Fit a linear regression model on a dataset. Interpret coefficients, make predictions, and evaluate performance using metrics like R-squared and MSE

Code :

Step 1: Import Libraries # Import required libraries

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score
```

Step 2: Create a Dataset and Save as CSV

 # Create a sample dataset

```
data = {

   'House_Size': [750, 800, 850, 900, 1000, 1100, 1200, 1300, 1400, 1500],

   'Price': [150000, 160000, 165000, 170000, 180000, 190000, 200000, 210000, 220000,
230000]

}
```

 # Convert the dataset into a DataFrame

```
df = pd.DataFrame(data)
```

 # Save to CSV file

```
df.to_csv('house_prices.csv',
index=False)
```

 # Display the dataset

```
print("Dataset:")
print(df)
```

Step 3: Load the Dataset

 # Load the dataset

dataset = pd.read_csv('house_prices.csv')
 # Display the first few

rows print("\nLoaded

Dataset:")

print(dataset.head())

Step 4: Split the Dataset into Training and Test Sets

 # Features and target variable

X = dataset[['House_Size']] **# Feature: House size**

y = dataset['Price'] **# Target: Price**

 # Split data into training and testing sets (80% train, 20% test)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
print("\nTraining and Testing Data Sizes:")

print("Training Data Size:", X_train.shape[0])

print("Testing Data Size:", X_test.shape[0])

Step 5: Fit a Linear Regression Model

 # Initialize and fit the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

 # Display the coefficients

print("\nModel Coefficients:")

print("Slope (m):", model.coef_[0])
print("Intercept (b):", model.intercept_)

Step 6: Make Predictions

 # Predict on the test set

y_pred = model.predict(X_test)
 # Display predictions
print("\nPredictions on Test Data:")
print("Actual Prices:", y_test.values)
print("Predicted Prices:", y_pred)

Step 7: Evaluate the Model

 # Calculate evaluation metrics

mse = mean_squared_error(y_test, y_pred) r2 = r2_score(y_test, y_pred)

 # Display metrics

print("\nModel Performance Metrics:")
print("Mean Squared Error (MSE):", mse)
print("R-squared (R^2):", r2)

Step 8: Visualize the Results # Scatter plot of the training data plt.scatter(X_train, y_train, color='blue', label='Training Data')

 # Plot the regression line

plt.plot(X_train, model.predict(X_train), color='red', label='Regression Line')
 # Scatter plot of the test data

plt.scatter(X_test, y_test, color='green', label='Test Data')

plt.title("Simple Linear Regression")

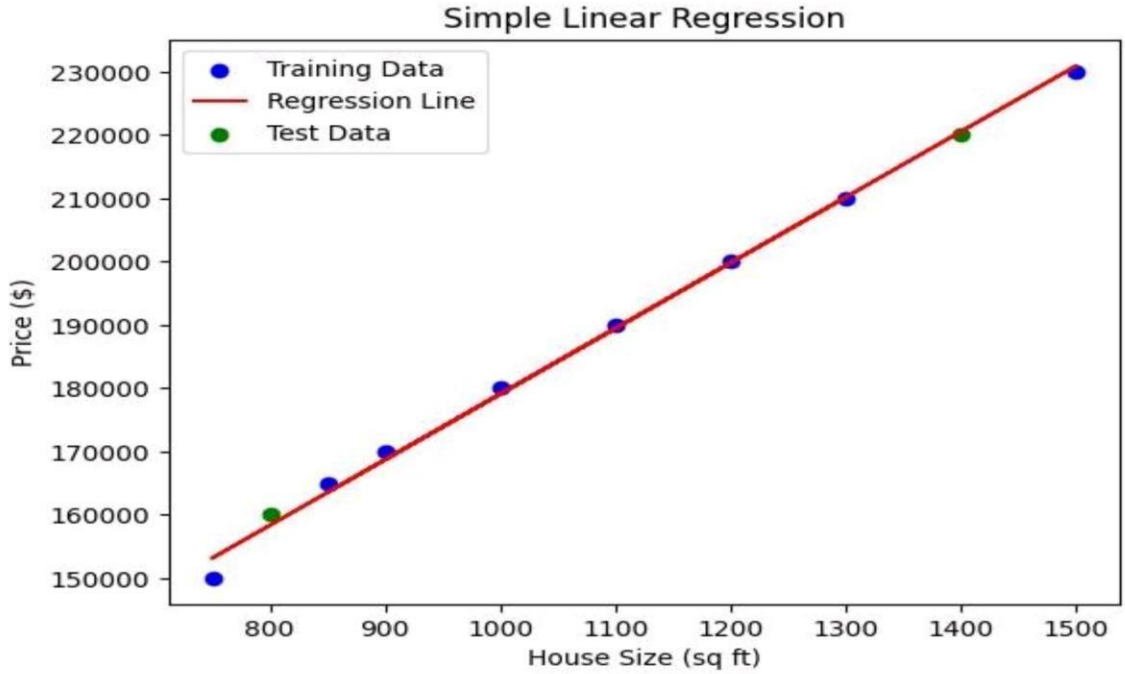
plt.xlabel("House Size (sq ft)")

plt.ylabel("Price ($)")

plt.legend()

plt.show()

Output :



3b. Multiple Linear Regression :

Extend linear regression to multiple feature. Handle feature selection and potential multicollinearity

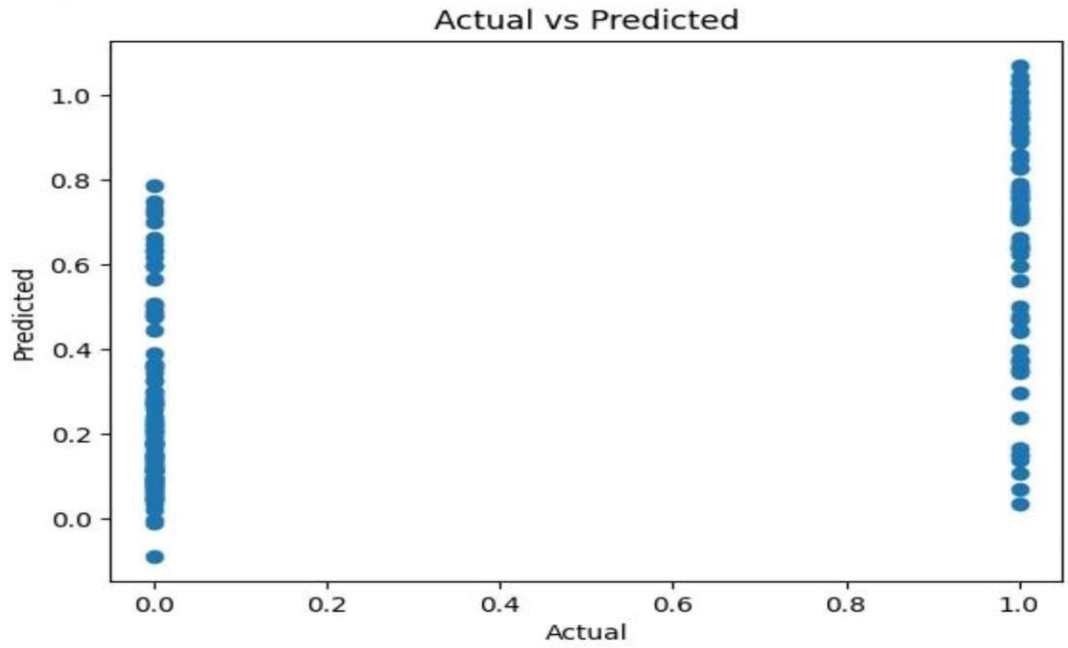
Code :

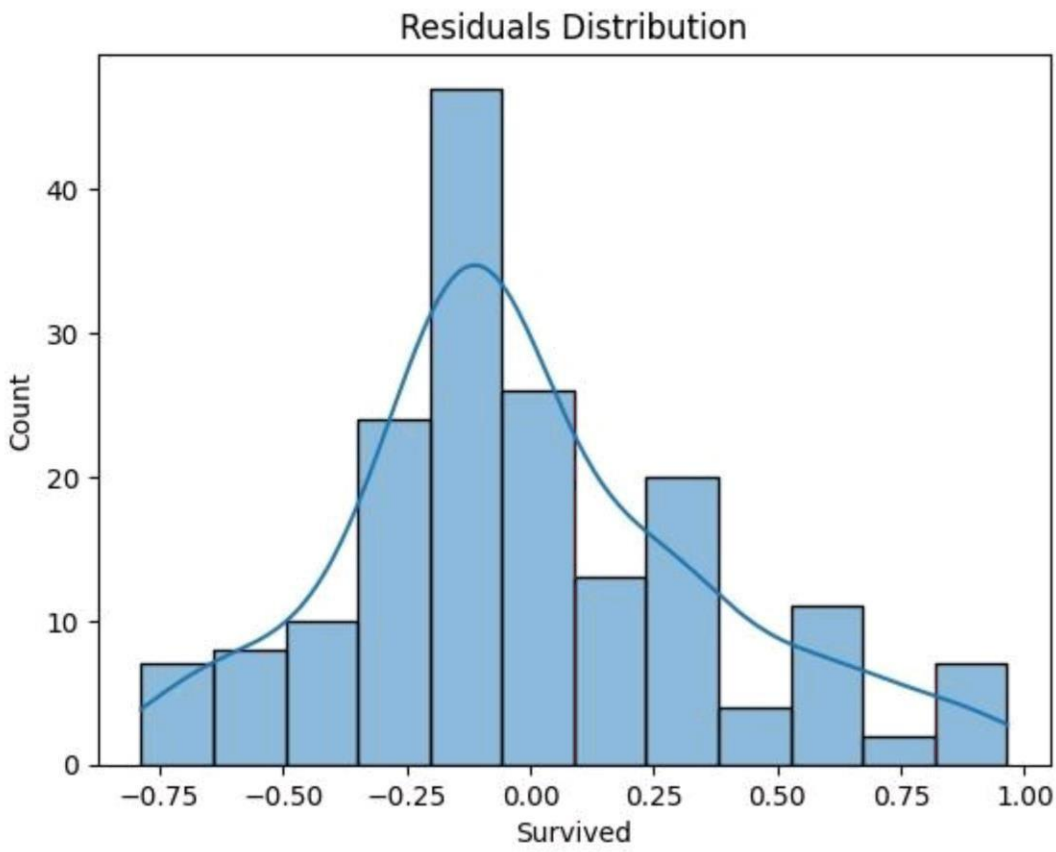
```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score

from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.preprocessing import LabelEncoder

# Import LabelEncoder

from sklearn.impute import SimpleImputer


# Load dataset

from google.colab import files uploaded = files.upload()  # Upload your CSV file


# Read the CSV file

data =pd.read_csv(list(uploaded.keys())[0])

# Display the first few rows
print(data.head())

# Check for null values and basic statistics
print(data.info())
print(data.describe())

# Define a function to calculate VIF
def calculate_vif(df):

# Select only numeric features for VIF calculation
numeric_df = df.select_dtypes(include=np.number)

# Drop rows with infinite or missing values
```

```
numeric_df = numeric_df.replace([np.inf, -np.inf], np.nan).dropna()
vif_data = pd.DataFrame()
vif_data["feature"] = numeric_df.columns
vif_data["VIF"] = [variance_inflation_factor(numeric_df.values, i)
for i in range(numeric_df.shape[1])]
   return vif_data

# Selecting features and target variable
X = data.drop("Survived", axis=1)
# Changed 'y' to 'Survived' y = data["Survived"]

# Handle categorical features (e.g., using Label Encoding)
for col in X.select_dtypes(include=['object']).columns:
   le = LabelEncoder()
   X[col] = le.fit_transform(X[col])

# Impute missing values using the mean (you can choose other strategies)
imputer = SimpleImputer(strategy='mean')
# Create an imputer instance
X = pd.DataFrame(imputer.fit_transform(X), columns=X.columns)
# Impute and update X

# Calculate VIF for initial features
print("VIF before handling multicollinearity:")
print(calculate_vif(X)) # Call the modified function

# Drop features based on VIF analysis (example: drop 'X1' if VIF is high)
# Check if the column exists before dropping
if 'X1' in X.columns:
   X = X.drop("X1", axis=1)  # Replace 'X1' with the actual high VIF feature name
else:
   print("Column 'X1' not found in the DataFrame.")

# Recalculate VIF
print("VIF after handling multicollinearity:")
print(calculate_vif(X))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and fit the model
model = LinearRegression()
model.fit(X_train, y_train)

# Get coefficients and intercept
print("Coefficients:", model.coef_)
print("Intercept:", model.intercept_)

# Predictions
y_pred = model.predict(X_test)

# Evaluation metrics
```

```
rmse = np.sqrt(mean_squared_error(y_test, y_pred)) r2 = r2_score(y_test, y_pred)
print(f"RMSE: {rmse}")
print(f"R^2: {r2}")
from sklearn.feature_selection import RFE

# Recursive Feature Elimination
rfe = RFE(estimator=LinearRegression(), n_features_to_select=5)

# Adjust features
rfe.fit(X_train, y_train)

# Selected features
print("Selected Features:", X.columns[rfe.support_])

# Scatter plot of actual vs predicted values
plt.scatter(y_test, y_pred) plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.title("Actual vs Predicted")
plt.show()

# Residuals
residuals = y_test - y_pred sns.histplot(residuals, kde=True)
plt.title("Residuals Distribution")
plt.show()
```

Output :




Residuals Distribution
Count
Survived
0
10
20
30
40
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

3c. Regualarized Linear Models :

Implement Regression variants like LASSO and Ridge on any generated dataset

Code :

1. **Set Up the Environment**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.datasets import make_regression
# Set random seed for reproducibility
np.random.seed(42)
```

2. **Generate a Synthetic Dataset**

```
# Generate synthetic data
X, y = make_regression(n_samples=1000,
# Number of samples
 n_features=10,
# Number of features
noise=15,
# Add some noise
 random_state=42
)
# Convert to DataFrame for exploration
data = pd.DataFrame(X, columns=[f"X{i}"
for i in range(1, 11)]) data["y"] = y
# Display the first few rows
```

```
print(data.head())
```

3. **Split the Dataset**

```
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data.drop("y", axis=1),
# Features
   data["y"],
# Target variable
 test_size=0.2,
 # 20% for testing
   random_state=42
)
```

4. **Train and Evaluate Ridge Regression**

```
# Initialize Ridge Regression with a regularization parameter (alpha)
ridge = Ridge(alpha=1.0)
# Train the model
ridge.fit(X_train, y_train)
# Predictions
ridge_pred = ridge.predict(X_test)
# Evaluate Ridge Regression
ridge_rmse = np.sqrt(mean_squared_error(y_test, ridge_pred))
ridge_r2 = r2_score(y_test, ridge_pred)
print(f"Ridge RMSE: {ridge_rmse}")
print(f"Ridge R^2: {ridge_r2}")
```

5. **Train and Evaluate Lasso Regression**

```
# Initialize Lasso Regression
lasso = Lasso(alpha=0.1)

# Train the model
lasso.fit(X_train, y_train)
```

 # Predictions

lasso_pred = lasso.predict(X_test)

 # Evaluate Lasso Regression

lasso_rmse = np.sqrt(mean_squared_error(y_test, lasso_pred))

lasso_r2 = r2_score(y_test, lasso_pred)

print(f"Lasso RMSE: {lasso_rmse}")

print(f"Lasso R^2: {lasso_r2}")

 # Features shrunk to

zero print("Lasso Coefficients:", lasso.coef_)

6. Train and Evaluate ElasticNet Regression

 # Initialize ElasticNet

elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5) # l1_ratio balances L1 and L2 penalties

 # Train the model

elastic_net.fit(X_train, y_train)

 # Predictions

elastic_net_pred = elastic_net.predict(X_test)

 # Evaluate

ElasticNet Regression elastic_net_rmse = np.sqrt(mean_squared_error(y_test, elastic_net_pred)) elastic_net_r2 = r2_score(y_test, elastic_net_pred)

print(f"ElasticNet RMSE: {elastic_net_rmse}")

print(f"ElasticNet R^2: {elastic_net_r2}")

7. Compare Results

 # Collect metrics

metrics = pd.DataFrame({

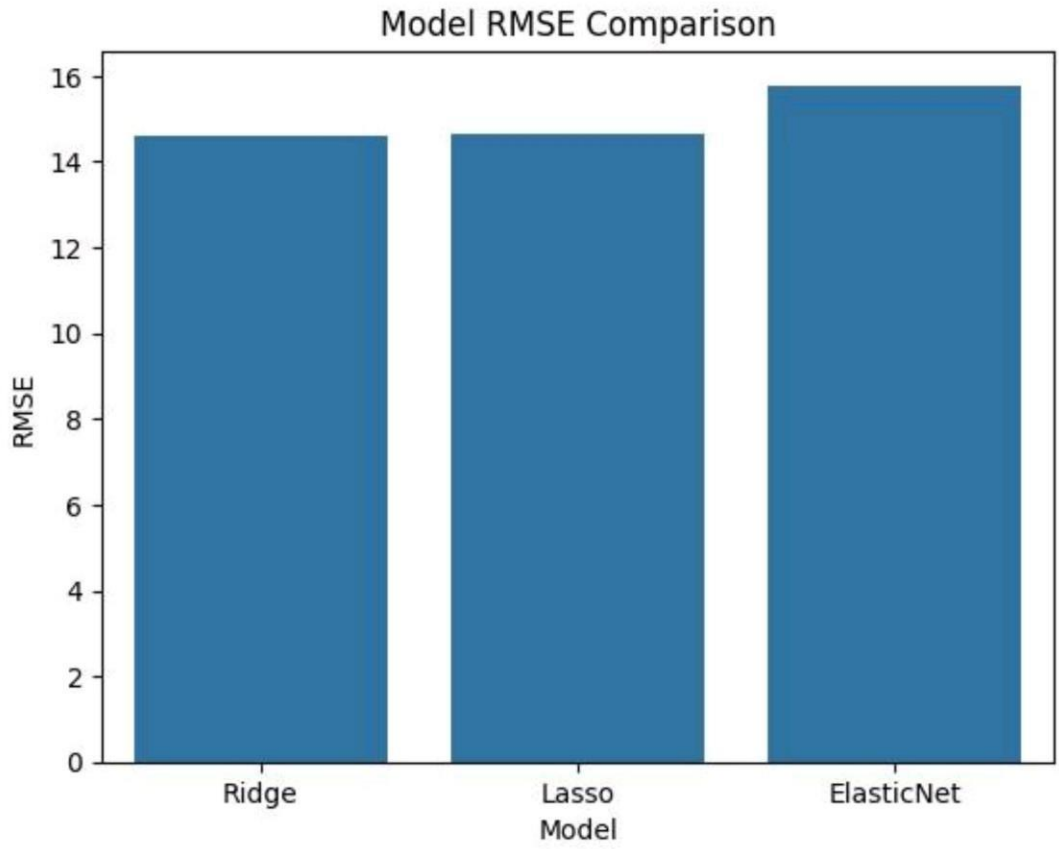
"Model": ["Ridge", "Lasso", "ElasticNet"],

"RMSE": [ridge_rmse, lasso_rmse, elastic_net_rmse],

"R^2": [ridge_r2, lasso_r2, elastic_net_r2]

```
})
print(metrics)
# Plot RMSE comparison
sns.barplot(data=metrics, x="Model", y="RMSE")
plt.title("Model RMSE Comparison")
plt.show()
```

Output :



Practical 4 : Discriminative Models

4a. Logistic Regression :

Perform binary classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve."

Code :

Step 1: Import Required Libraries

 # Import necessary libraries

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, roc_curve, auc
import matplotlib.pyplot as plt
```

Step 2: Prepare the Dataset

```
from sklearn.datasets import make_classification
```

 # Create a synthetic dataset

```
X, y = make_classification(n_samples=1000, n_features=10, n_classes=2, random_state=42)
```

 # Split data into training and testing sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

Step 3: Train the Logistic Regression Model

 # Initialize the logistic regression model

```
logreg = LogisticRegression()
```

 # Train the model on the training data

```
logreg.fit(X_train, y_train)
```

Step 4: Make Predictions

 # Predict labels for the test set

```
y_pred = logreg.predict(X_test)
```

 # Predict probabilities for the ROC curve

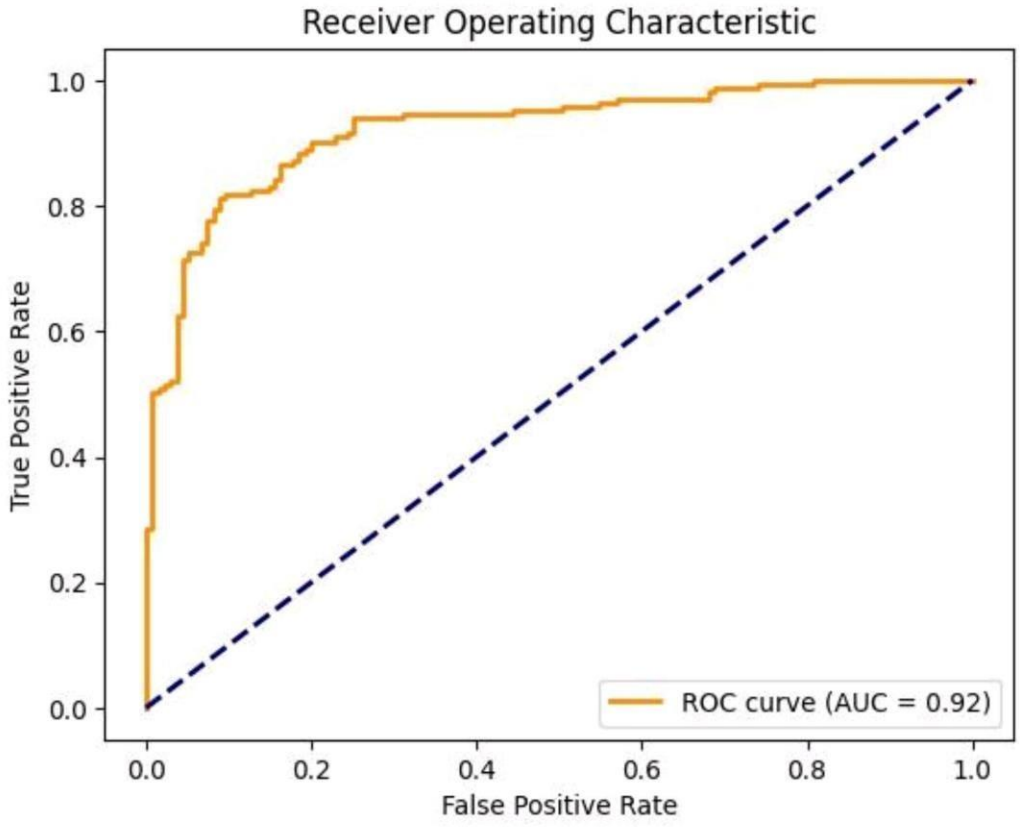
```
y_prob = logreg.predict_proba(X_test)[:, 1]
```

Step 5: Evaluate the Model

 # Calculate metrics

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

Output :



4b .Implement and demonstrate k-nearest Neighbor algorithm. Read the training data from a .CSV file and build the model to classify a test sample. Print both correct and wrong predictions.

Code :

Step 1: Import Required Libraries # Import necessary libraries

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

from google.colab import files
```

Step 2: Create or Upload the CSV File

```
# Check if the user wants to create a dataset or upload one
print("Do you have a CSV file to upload? (yes/no)")
response = input().lower()

if response == "yes":

   uploaded = files.upload()
filename = list(uploaded.keys())[0]
else:

 # Create a synthetic dataset

from sklearn.datasets import make_classification

 # Generate synthetic data

X,y=make_classification(n_samples=200,n_features=5, n_classes=2, random_state=42)


# Combine features and target into a single DataFrame

data = pd.DataFrame(X, columns=[f"Feature_{i}"

for i in range(X.shape[1])])

   data['Target'] = y

# Save the dataset to a CSV file

filename = "synthetic_data.csv"

data.to_csv(filename, index=False)

print(f"Synthetic dataset saved as {filename}.")
```

Step 3: Load the CSV File into a DataFrame

```
# Load the dataset into a DataFrame
data = pd.read_csv(filename)
# Display the first few rows of the dataset
print("Loaded Dataset:")
print(data.head())
```

Step 4: Preprocess the Data

 # Separate features (X) and labels (y)

```
X = data.iloc[:, :-1].values  # All columns except the last one
y = data.iloc[:, -1].values   # Last column as the target
```

 # Split the dataset into training and testing sets (80% train, 20% test)

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 5: Train the k-NN Model #

Initialize the k-NN model with k=3 knn

```
= KNeighborsClassifier(n_neighbors=3) #
```

Train the model on the training data

```
knn.fit(X_train, y_train)
```

Step 6: Predict Test Samples #

Predict the labels for the test set

```
y_pred = knn.predict(X_test)
```

Step 7: Evaluate and Print Predictions #
Calculate and display the accuracy

```
accuracy = accuracy_score(y_test, y_pred)
print(f"\nModel Accuracy:
{accuracy:.2f}\n")
```

 # Display correct and incorrect predictions

```
print("Correct Predictions:")
for i in range(len(y_test)):
    if y_pred[i] == y_test[i]:
print(f"Sample {i}: Predicted={y_pred[i]}, Actual={y_test[i]}")
print("\nIncorrect Predictions:")

for i in range(len(y_test)):
```

```
        if y_pred[i] != y_test[i]:
print(f"Sample {i}: Predicted={y_pred[i]}, Actual={y_test[i]}")
```

Output :

```
[ ]
    Model Accuracy: 0.88

    Correct Predictions:
    Sample 0: Predicted=0, Actual=0
    Sample 1: Predicted=1, Actual=1
    Sample 2: Predicted=1, Actual=1
    Sample 3: Predicted=0, Actual=0
    Sample 4: Predicted=1, Actual=1
    Sample 5: Predicted=1, Actual=1
    Sample 6: Predicted=0, Actual=0
    Sample 7: Predicted=0, Actual=0
    Sample 9: Predicted=1, Actual=1
    Sample 10: Predicted=1, Actual=1
    Sample 11: Predicted=1, Actual=1
    Sample 12: Predicted=0, Actual=0
    Sample 13: Predicted=0, Actual=0
    Sample 14: Predicted=0, Actual=0
    Sample 15: Predicted=0, Actual=0
    Sample 16: Predicted=0, Actual=0
    Sample 17: Predicted=1, Actual=1
    Sample 18: Predicted=1, Actual=1
    Sample 19: Predicted=0, Actual=0
    Sample 20: Predicted=0, Actual=0
    Sample 22: Predicted=1, Actual=1
    Sample 23: Predicted=1, Actual=1
    Sample 24: Predicted=1, Actual=1
    Sample 25: Predicted=1, Actual=1
    Sample 26: Predicted=1, Actual=1
    Sample 27: Predicted=0, Actual=0
    Sample 28: Predicted=0, Actual=0
    Sample 30: Predicted=1, Actual=1
    Sample 31: Predicted=1, Actual=1
    Sample 32: Predicted=1, Actual=1
    Sample 34: Predicted=0, Actual=0
    Sample 35: Predicted=1, Actual=1
    Sample 36: Predicted=1, Actual=1
    Sample 38: Predicted=1, Actual=1
    Sample 39: Predicted=1, Actual=1
```

```
    Incorrect Predictions:
    Sample 8: Predicted=1, Actual=0
    Sample 21: Predicted=1, Actual=0
    Sample 29: Predicted=0, Actual=1
    Sample 33: Predicted=1, Actual=0
    Sample 37: Predicted=1, Actual=0
```

4c. Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree.

Code :

Step 1: Import Required Libraries

 # Import necessary libraries

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor,
plot_tree

from sklearn.metrics import accuracy_score, mean_squared_error

import matplotlib.pyplot as plt

from google.colab import files
```

Step 2: Create or Upload the CSV File

```
# Check if the user wants to upload a file or generate one
print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

    # Upload the CSV file

     uploaded = files.upload()

     filename = list(uploaded.keys())[0]

else:

  # Generate synthetic data (classification or regression)

   from sklearn.datasets import make_classification, make_regression
print("Choose a task: (1) Classification (2) Regression")

task = int(input())

if task == 1:

  # Generate synthetic classification data

     X, y = make_classification(n_samples=200, n_features=5, random_state=42)

     task_type = "classification"
```

```
    else:
        # Generate synthetic regression data
        X, y = make_regression(n_samples=200, n_features=5, random_state=42)
        task_type = "regression"
    # Combine features and target into a single DataFrame
data = pd.DataFrame(X, columns=[f"Feature_{i}"
for i in range(X.shape[1])]) data['Target'] = y
    # Save the dataset to a CSV file
filename = "synthetic_data.csv"
data.to_csv(filename, index=False)
print(f"Synthetic {task_type} dataset saved as {filename}.")
```

Step 3: Load the Dataset

```
# Load the dataset
data =pd.read_csv(filename)
# Display the first few rows of the dataset
print("Dataset Preview:")
print(data.head())
```

Step 4: Preprocess the Data

```
# Separate features and target
X = data.iloc[:, :-1].values  # All columns except the last one
y = data.iloc[:, -1].values   # Last column as the target
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 5: Build the Decision Tree

```
# Define the tree depth to avoid overfitting
max_depth = 3
```

 # Initialize the model

```
if task_type =="classification":
   model = DecisionTreeClassifier(max_depth=max_depth, random_state=42)
else:
   model = DecisionTreeRegressor(max_depth=max_depth, random_state=42)
```

 # Train the model

```
model.fit(X_train, y_train)
```

Step 6: Make Predictions

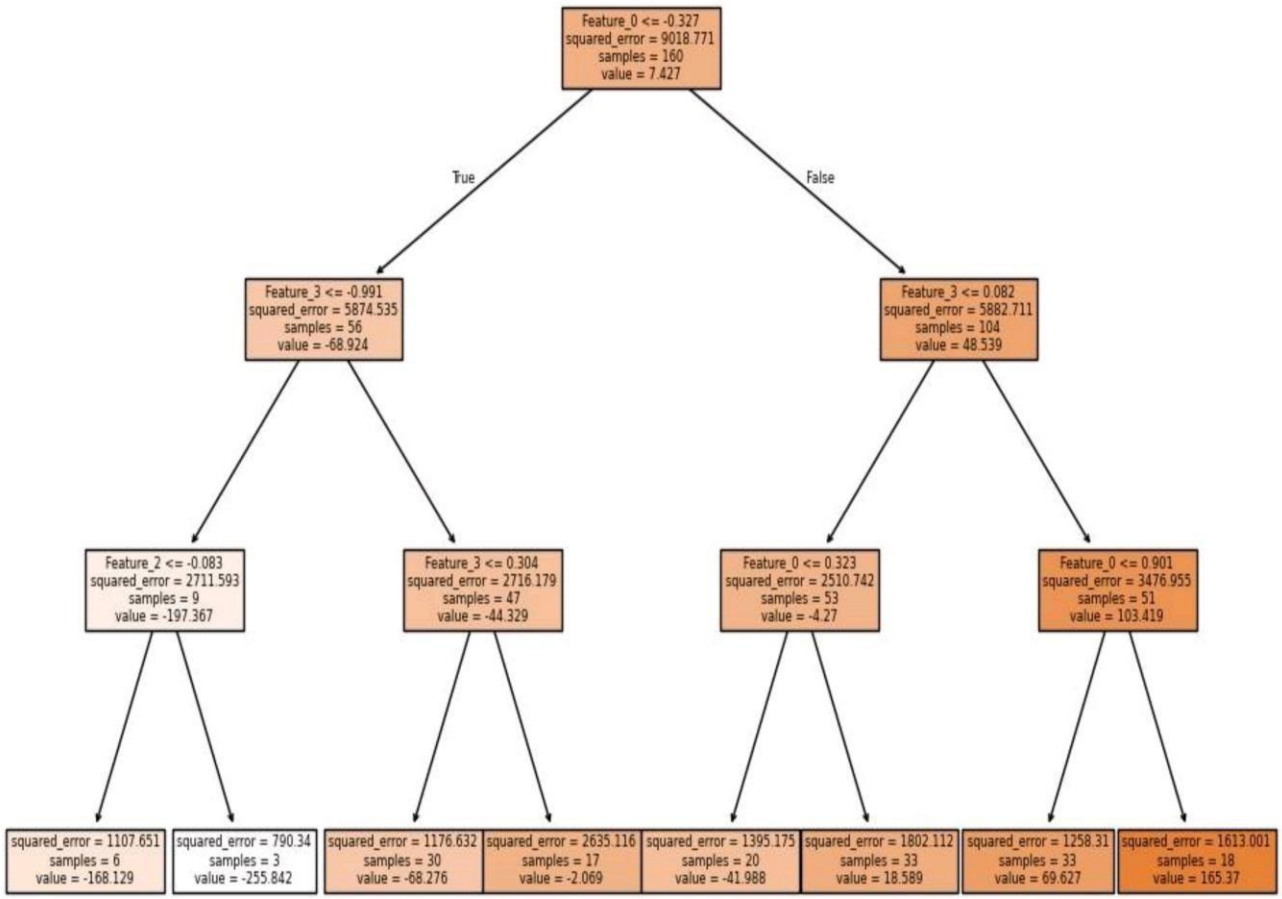
```
# Predict on the test set
y_pred = model.predict(X_test)
# Evaluate the model
if task_type == "classification":
   accuracy = accuracy_score(y_test, y_pred)
   print(f"Accuracy: {accuracy:.2f}")
else:
   mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")
```

Step 7: Visualize the Tree

 # Visualize the decision tree

```
plt.figure(figsize=(12, 8))
plot_tree(model, feature_names=data.columns[:-1], class_names=str(np.unique(y))
if task_type == "classification" else None, filled=True)
plt.title("Decision Tree Visualization")
plt.show()
```

Output :


Decision Tree Visualization
Feature_0 <= -0.327
squared_error = 9018.771
samples = 160
value = 7.427
True
False
Feature_3 <= -0.991
squared_error = 5874.535
samples = 56
value = -68.924
Feature_3 <= 0.082
squared_error = 5882.711
samples = 104
value = 48.539
Feature_2 <= -0.083
squared_error = 2711.593
samples = 9
value = -197.367
Feature_3 <= 0.304
squared_error = 2716.179
samples = 47
value = -44.329
Feature_0 <= 0.323
squared_error = 2510.742
samples = 53
value = -4.27
Feature_0 <= 0.901
squared_error = 3476.955
samples = 51
value = 103.419
squared_error = 1107.651
samples = 6
value = -168.129
squared_error = 790.34
samples = 3
value = -255.842
squared_error = 1176.632
samples = 30
value = -68.276
squared_error = 2635.116
samples = 17
value = -2.069
squared_error = 1395.175
samples = 20
value = -41.988
squared_error = 1802.112
samples = 33
value = 18.589
squared_error = 1258.31
samples = 33
value = 69.627
squared_error = 1613.001
samples = 18
value = 165.37

4d. Implement a Support Vector Machine for any relevant dataset.

Code:

Step 1: Import Required Libraries

 # Import necessary libraries

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score, classification_report
from google.colab import files
```

Step 2: Create or Upload a Dataset

 # Check if the user wants to upload a file or generate one

```
print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":
```

 # Upload the CSV file

```
uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:
```

 # Generate synthetic classification data

```
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=200, n_features=5, n_classes=2, random_state=42)
```

 # Combine features and target into a DataFrame

```
data = pd.DataFrame(X, columns=[f"Feature_{i}"

for i in range(X.shape[1])])

   data['Target'] = y
```

#Save the synthetic dataset to a CSV file

```
filename="synthetic_data.csv"
data.to_csv(filename,index=False)
print(f"Synthetic dataset saved as {filename}.")
```

Step 3: Load the Dataset

 # Load the dataset into a DataFrame

```
data = pd.read_csv(filename)
```

 # Display the first few rows of the dataset

```
print("Dataset Preview:")
print(data.head())
```

Step 4: Preprocess the Data

 # Separate features (X) and target (y)

```
X = data.iloc[:, :-1].values  # All columns except the last one
y = data.iloc[:, -1].values   # Last column as the target
```

 # Split the dataset into training (80%) and testing (20%) sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 5: Train the Support Vector Machine

 # Initialize the SVM model (use RBF kernel as default)

```
svm_model = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42)
```

 # Train the SVM model on the training data

```
svm_model.fit(X_train, y_train)
```

Step 6: Make Predictions

 # Predict the labels for the test set

```
y_pred = svm_model.predict(X_test)
```

Step 7: Evaluate the Model

 # Calculate and print the accuracy

```
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")
```

 # Print a detailed classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

Step 8: Visualize the Decision Boundary (Optional for 2D

Data)

import matplotlib.pyplot as plt

 # Generate 2D synthetic data

from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=100, centers=2, random_state=42, cluster_std=1.5)

 # Fit the SVM on this data

svm_model.fit(X, y)

#Plot the decision boundary
plt.figure(figsize=(8, 6))

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolor='k')
 # Create a grid to evaluate the model

xx, yy = np.meshgrid (np.linspace(X[:, 0].min(), X[:, 0].max(), 100), np.linspace(X[:, 1].min(), X[:, 1].max(), 100))
Z = svm_model.decision_function(np.c_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

 # Plot the decision boundary and margins

plt.contour(xx, yy, Z, levels=[-1, 0, 1], linestyles=['--', '-', '--'], colors='k')

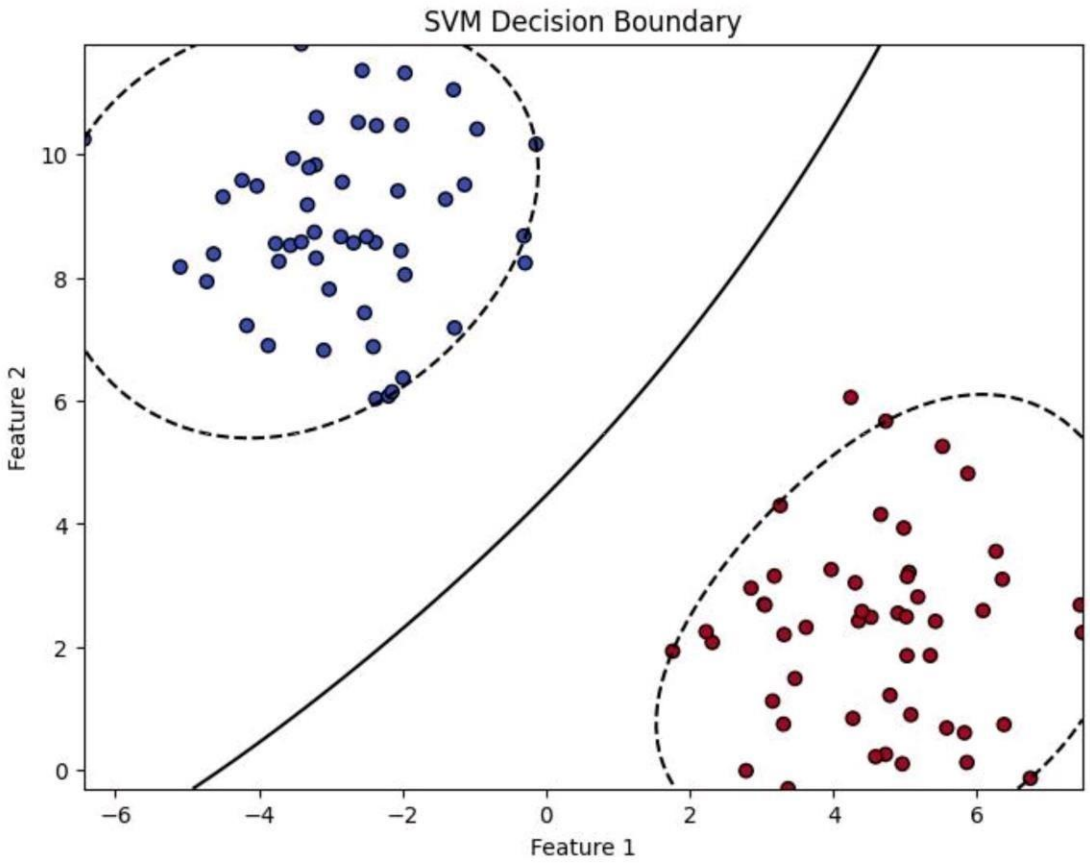
plt.title("SVM Decision Boundary")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

Output :


SVM Decision Boundary
Feature 2
Feature 1
10
8
6
4
2
0
−6
−4
−2
0
2
4
6

4e. Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree.

Code :

Step 1: Import Required Libraries # Import necessarylibraries

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score, classification_report
from google.colab import files
```

Step 2: Create or Upload a Dataset

```
# Check if the user wants to upload a file or generate one
print("Do you have a CSV file to upload? (yes/no)")
response = input().lower()
if response == "yes":
# Upload the CSV file
uploaded = files.upload()
filename = list(uploaded.keys())[0]
else:
# Generate synthetic classification data

from sklearn.datasets import make_classification

 X, y = make_classification(n_samples=300, n_features=10, n_classes=2, random_state=42)

 # Combine features and target into a DataFrame

data = pd.DataFrame(X, columns=[f"Feature_{i}"

for i in range(X.shape[1])])

    data['Target'] = y

# Save the synthetic dataset to a CSV file
filename = "synthetic_data.csv"
data.to_csv(filename, index=False)
print(f"Synthetic dataset saved as {filename}.")
```

Step 3: Load the Dataset

 # Load the dataset

```
data =pd.read_csv(filename)

# Display the first few rows of the dataset
print("Dataset Preview:")
print(data.head())
```

Step 4: Preprocess the Data

```
# Separate features (X) and target (y)

X = data.iloc[:, :-1].values  # All columns except the last one
y = data.iloc[:, -1].values   # Last column as the target

# Split the dataset into training (80%) and testing (20%) sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 5: Train a Single Decision Tree Classifier

```
# Initialize and train the Decision Tree model
decision_tree =DecisionTreeClassifier(random_state=42)
decision_tree.fit(X_train, y_train)
# Predict and evaluate
y_pred_tree = decision_tree.predict(X_test)
accuracy_tree = accuracy_score(y_test, y_pred_tree)
print(f"Decision Tree Accuracy: {accuracy_tree:.2f}")
```

Step 6: Train a Random Forest Classifier

```
# Initialize the Random Forest model with hyperparameter tuning

random_forest = RandomForestClassifier(n_estimators=100, max_features='sqrt',
random_state=42)
# Train the model
random_forest.fit(X_train, y_train)

# Predict and evaluate

y_pred_rf = random_forest.predict(X_test)

accuracy_rf = accuracy_score(y_test, y_pred_rf)

print(f"Random Forest Accuracy (100 trees, sqrt features): {accuracy_rf:.2f}")
```

Step 7: Experiment with Random Forest Hyperparameters

```
# Experiment with fewer trees and different feature sampling
rf_experiment = RandomForestClassifier(n_estimators=50, max_features=3,
random_state=42)

rf_experiment.fit(X_train, y_train)
```

 # Predict and evaluate

y_pred_rf_exp = rf_experiment.predict(X_test)

accuracy_rf_exp = accuracy_score(y_test, y_pred_rf_exp)

print(f"Random Forest Accuracy (50 trees, max_features=3): {accuracy_rf_exp:.2f}")

Step 8: Compare the Models

print("\nModel Comparison:")

print(f"Decision Tree Accuracy: {accuracy_tree:.2f}")

print(f"Random Forest Accuracy (100 trees): {accuracy_rf:.2f}")

print(f"Random Forest Accuracy (50 trees, max_features=3): {accuracy_rf_exp:.2f}")

Step 9: Visualize Feature Importance (Optional)

import matplotlib.pyplot as plt

 # Extract feature importance from the Random Forest model

feature_importances = random_forest.feature_importances_

 # Plot the feature importance

plt.figure(figsize=(10, 6))

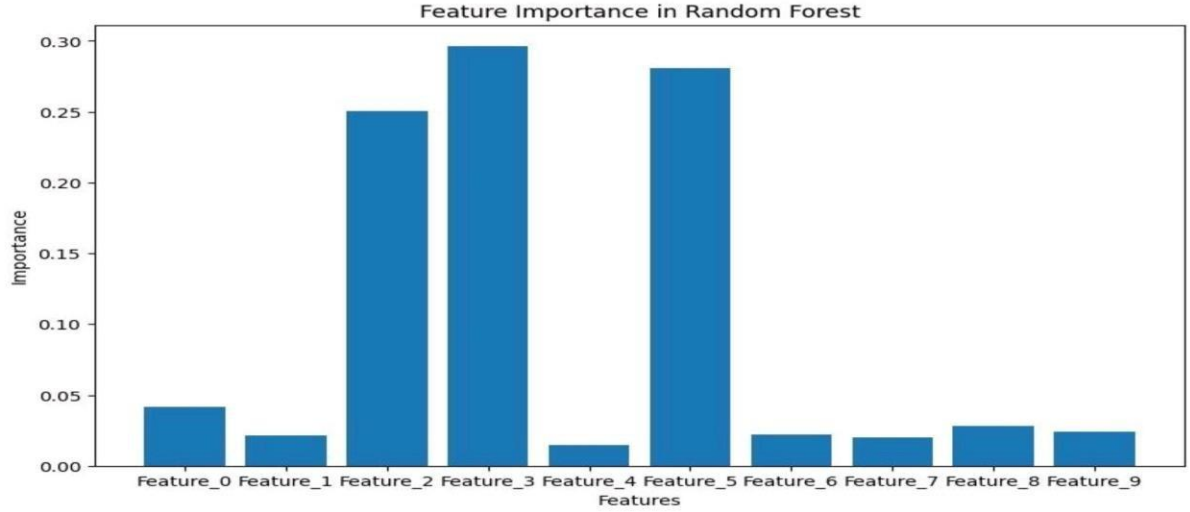
plt.bar(range(len(feature_importances)), feature_importances, tick_label=data.columns[:-1])
plt.title("Feature Importance in Random Forest")

plt.xlabel("Features")

plt.ylabel("Importance")

plt.show()

Output :



4f. Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters and explore feature importance.

Code :

Step 1: Import Required Libraries

 # Import necessary libraries

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.metrics import accuracy_score, classification_report
from xgboost import XGBClassifier, plot_importance

import matplotlib.pyplot as plt

from google.colab import files
```

Step 2: Create or Upload a Dataset

 # Check if the user wants to upload a file or generate

one print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

Upload the CSV file

uploaded=files.upload()

filename = list(uploaded.keys())[0]

else:

 # Generate synthetic classification data

from sklearn.datasets import make_classification

X, y = make_classification(n_samples=300, n_features=10, n_classes=2, random_state=42)

 # Combine features and target into a DataFrame

data = pd.DataFrame(X, columns=[f"Feature_{i}"

for i in range(X.shape[1])])data['Target'] = y

 # Save the synthetic dataset to a CSV file

```
filename="synthetic_data.csv"

data.to_csv(filename,index=False)

print(f"Synthetic dataset saved as {filename}.")
```

Step 3: Load the Dataset

 # Load the dataset

```
data = pd.read_csv(filename)
```
 # Display the first few rows of the dataset print("Dataset Preview:")
print(data.head())

Step 4: Preprocess the Data

 # Separate features (X) and target (y)

```
X = data.iloc[:, :-1].values  # All columns except the last one
y = data.iloc[:, -1].values   # Last column as the target
```

 # Split the dataset into training (80%) and testing (20%) sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 5: Train a Basic XGBoost Model

 # Initialize and train the XGBoost model with default parameters

```
xgb = XGBClassifier(random_state=42)

xgb.fit(X_train, y_train)
```

 # Predict and evaluate the model

```
y_pred = xgb.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

print(f"XGBoost Accuracy (Default Parameters): {accuracy:.2f}")
```
Step 6: Tune Hyperparameters with GridSearchCV

 # Define a grid of hyperparameters
```
param_grid = { 'n_estimators': [50, 100, 150],'learning_rate': [0.01, 0.1, 0.2], 'max_depth': [3, 5, 7] }
```
 # Initialize GridSearchCV

```
grid_search =
GridSearchCV(estimator=XGBClassifier(random_state=42),param_grid=param_grid,
scoring='accuracy', cv=3,verbose=1)
```
 # Fit the model with grid search
```
grid_search.fit(X_train, y_train)
```

 # Best parameters from GridSearch

print(f"Best Parameters: {grid_search.best_params_}")
 # Train the final model with the best parameters
best_xgb = grid_search.best_estimator_
 # Predict and evaluate

y_pred_best = best_xgb.predict(X_test)

accuracy_best = accuracy_score(y_test, y_pred_best)
print(f"XGBoost Accuracy (Tuned Parameters): {accuracy_best:.2f}")

Step 7: Explore Feature Importance

 # Plot feature importance for the tuned model

plt.figure(figsize=(10, 6))

plot_importance(best_xgb,importance_type='weight', xlabel="Importance",

ylabel="Features")

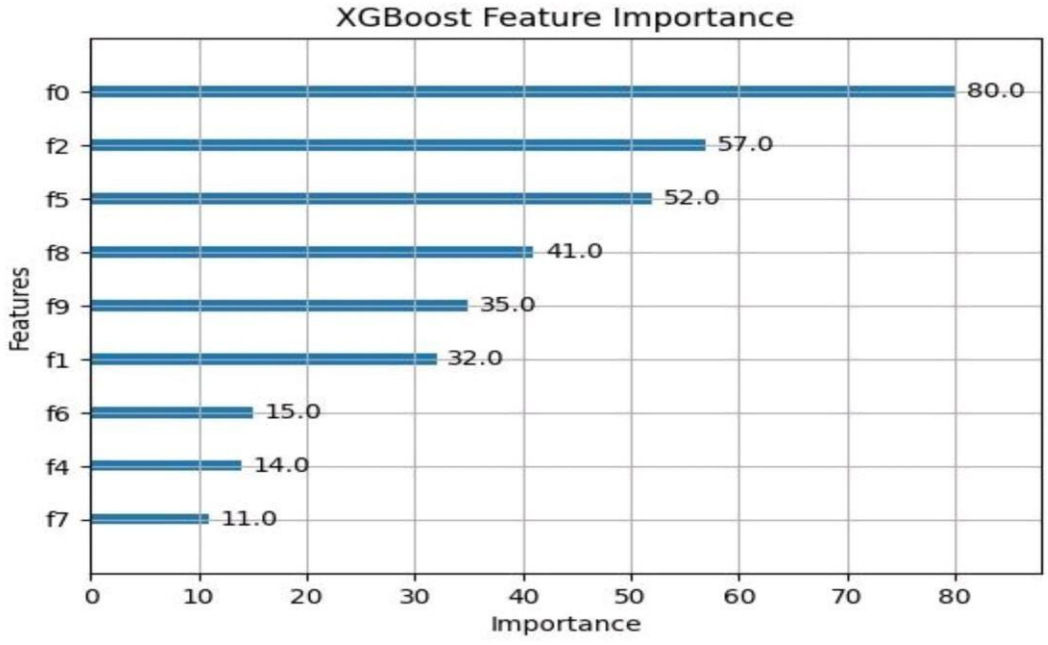
plt.title("XGBoost Feature Importance")

plt.show()

Step 8: Evaluate the Model

 # Print a detailed classification report
print("Classification Report:")
print(classification_report(y_test, y_pred_best))

Output :



Practical 5

5a. Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample.

Step 1: Import Required Libraries

 # Import necessary libraries

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score,classification_report
from sklearn.naive_bayes import GaussianNB

from google.colab import files
```

Step 2: Create or Upload a Dataset

 # Ask if the user wants to upload a file or generate one

```
print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":
```

 # Upload the CSV file

```
uploaded =files.upload()

 filename = list(uploaded.keys())[0]

else:
```

 # Generate synthetic classification data

```
from sklearn.datasets import make_classification

   X, y = make_classification(n_samples=300, n_features=8, n_classes=2, random_state=42)
```

 # Combine features and target into a DataFrame

```
data = pd.DataFrame(X, columns=[f"Feature_{i}" for i in range(X.shape[1])])
 data['Target'] = y
```

 # Save the synthetic dataset to a CSV file

```
filename="synthetic_naive_bayes_data.csv"

data.to_csv(filename,index=False)

print(f"Synthetic dataset saved as {filename}.")
```

Step 3: Load the Dataset

 # Load the dataset

```
data =pd.read_csv(filename)
```

 # Display the first few rows of the dataset

```
print("Dataset Preview:")

print(data.head())
```

Step 4: Preprocess the Data

 # Separate features (X) and target (y)

```
X = data.iloc[:, :-1].values  # All columns except the last one
y = data.iloc[:, -1].values   # Last column as the target
```

 # Split the dataset into training (80%) and testing (20%) sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 5: Train a Naive Bayes Classifier

 # Initialize the Gaussian Naive Bayes classifier

```
naive_bayes = GaussianNB()
```

 # Train the model

```
naive_bayes.fit(X_train, y_train)
```

Step 6: Make Predictions and Evaluate

 # Predict on the test set

```
y_pred =naive_bayes.predict(X_test)
```

 # Evaluate the model

```
accuracy = accuracy_score(y_test, y_pred)
print(f"Naive Bayes Accuracy:
{accuracy:.2f}")
```

 # Detailed classification report

```
print("Classification Report:")

print(classification_report(y_test, y_pred))
```

Step 7: Test the Model with a Custom Sample

 # Define a sample test input (replace with meaningful values based on your dataset)

test_sample = [X_test[0]]

 # Taking the first test sample for demonstration

 # Predict the class for the test sample

predicted_class = naive_bayes.predict(test_sample)

print(f"Test Sample: {test_sample}")

print(f"Predicted Class: {predicted_class[0]}")

Output :

```
Dataset Preview:
   Feature_0  Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  \
0  -1.274158   1.317988  -2.423879   0.906946  -1.583903  -0.331811
1   1.607963  -1.649959   0.299293  -0.891720   1.301741   1.508502
2  -0.154167   0.161033   2.210523   0.139400  -0.557492   0.087713
3  -0.920991   0.949136  -1.613561   0.588410   1.471170  -0.529287
4   1.013304  -1.038578  -0.305225  -0.539334  -0.609512   1.048078

   Feature_6  Feature_7  Target
0  -0.452306   0.760415       1
1   0.742095   1.561511       0
2   0.963879  -1.369803       0
3  -1.371901  -0.209324       0
4  -1.065114  -0.186971       0
```

```
Test Sample: [array([-0.90320608,  0.9220511 , -1.32308979,  0.41081065,  1.64201516,
       -1.23559176, -0.63896175,  1.00981709])]
Predicted Class: 1
```

5b. Implement Hidden Markov Models using hmmlearn

Code :

Step 1: Install Required Libraries

 # Install hmmlearn

```
!pip install hmmlearn
```

Step 2: Import Required Libraries
 # Import necessary libraries

```
import numpy as np

import pandas as pd

from hmmlearn import hmm

import matplotlib.pyplot as plt
```

Step 3: Create or Load a Dataset

 # Generate synthetic observable data

```
np.random.seed(42)
```

 # Create a sequence of observations and hidden states

```
observations = np.random.choice(['A', 'B', 'C'], size=100, p=[0.5, 0.3,
0.2])

hidden_states = np.random.choice(['X', 'Y'], size=100, p=[0.6, 0.4])
```

 # Save the data in a DataFrame for analysis

```
data = pd.DataFrame({'Observations': observations, 'Hidden States': hidden_states})
print("Generated Data:")

print(data.head())
```

Step 4: Encode Observations

 # Encode the observations into integers

```
observation_mapping = {obs: idx for idx, obs in enumerate(np.unique(observations))}
encoded_observations = np.array([observation_mapping[obs] for obs in observations])
```

 # Print the mapping

```
print("Observation Encoding:")

print(observation_mapping)
```

Step 5: Initialize and Configure the HMM

```
# Initialize the HMM model
n_states = 2  # Number of hidden states
n_observations = len(observation_mapping)
# Number of unique observations
model = hmm.MultinomialHMM(n_components=n_states, random_state=42, n_iter=100,
tol=0.01)

# Define start probabilities (initial distribution of states)
start_probs = np.array([0.6, 0.4])  # Assumed probabilities
model.startprob_ = start_probs

# Define transition probabilities between states
trans_probs = np.array([
   [0.7, 0.3],  # From state X
   [0.4, 0.6],  # From state Y])
model.transmat_ = trans_probs

# Define emission probabilities (probability of observations given states)
emission_probs = np.array([
   [0.5, 0.4, 0.1],  # State X emits A, B, C
   [0.2, 0.3, 0.5],  # State Y emits A, B, C
])
model.emissionprob_ = emission_probs

# Print the configured model parameters
print("Start Probabilities:", model.startprob_)
print("Transition Matrix:", model.transmat_)
print("Emission Probabilities:",
model.emissionprob_)
```

Step 6: Train the Model

```
# Reshape the data for HMM (requires 2D array)
encoded_observations = encoded_observations.reshape(-1, 1)

# Fit the model
model.fit(encoded_observations)

# Predict hidden states for the observations
predicted_states = model.predict(encoded_observations)
```

 # Print the predicted states
print("Predicted States:")
print(predicted_states)

Step 7: Visualize the Results

 # Map predicted states back to their original labels

state_mapping = {0: 'X', 1: 'Y'}

predicted_state_labels = [state_mapping[state] for state in predicted_states]

 # Add predicted states to the DataFrame
data['Predicted States'] = predicted_state_labels

 # Display the first few rows with predicted states
print("Data with Predicted States:")
print(data.head())
 # Plot the observations and predicted states

plt.figure(figsize=(12, 6))

plt.plot(data['Observations'], label='Observations', marker='o', linestyle='-', alpha=0.7)
plt.plot(data['Predicted States'], label='Predicted States', marker='x', linestyle='--', alpha=0.7)
plt.legend()

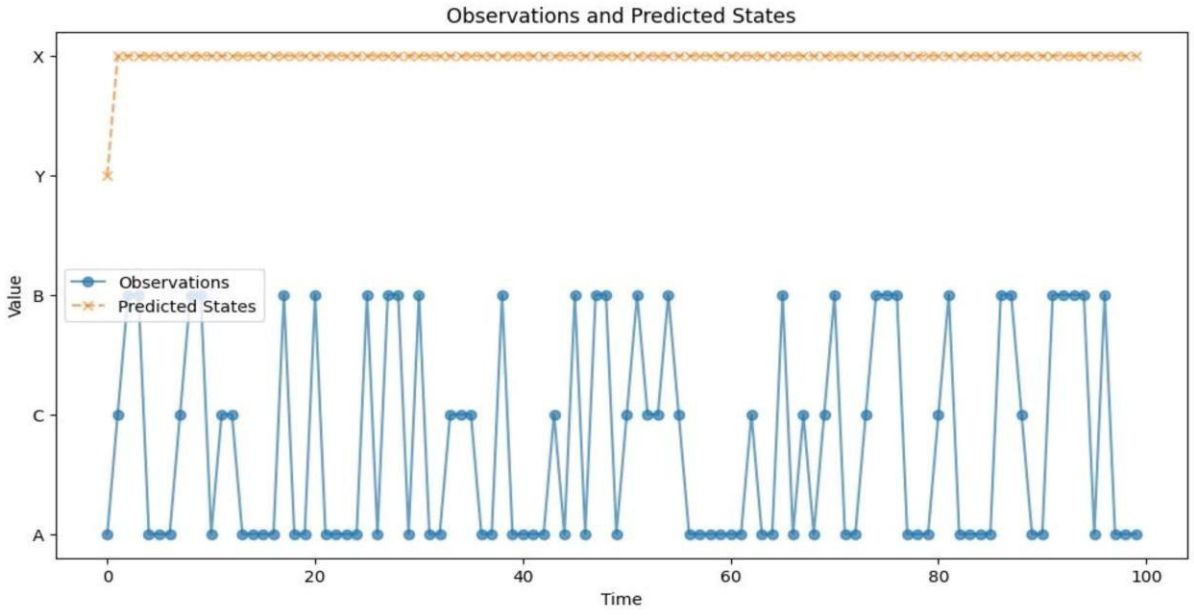
plt.title("Observations and Predicted States")

plt.xlabel("Time")

plt.ylabel("Value")

plt.show()

Output :



Practical 6 : Probabilistic Model

6a. Implement Bayesian Linear Regression to explore prior and posterior distribution.

Bayesian Linear Regression is a probabilistic approach to linear regression that incorporates uncertainty in the model parameters. Instead of estimating point values for parameters (as in traditional linear regression), we estimate distributions over the parameters.

Code :

Step 1: Install Required Libraries

 # Install necessary libraries

```
!pip install matplotlib seaborn scikit-learn
```

Step 2: Import Required Libraries

 # Import necessary libraries

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.linear_model import BayesianRidge
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from google.colab import files
```

Step 3: Create or Upload a Dataset

 # Upload a CSV file if you have one

```
print("Do you have a CSV file to upload? (yes/no)")
response = input().lower()

if response == "yes":

# Upload the CSV file

    uploaded = files.upload()
```

```
    filename = list(uploaded.keys())[0]
else:
 # Generate synthetic data for demonstration
np.random.seed(42)
   X = np.random.rand(100, 1) * 10
# Random data between 0 and 10
y = 2 * X + 1 + np.random.randn(100, 1) * 2
# y = 2x + 1 with some noise
# Convert to a DataFrame
data = pd.DataFrame(np.hstack((X, y)), columns=["X", "y"])
# Save to CSV for convenience
filename="synthetic_data.csv"
data.to_csv(filename,index=False)
print(f"Synthetic dataset saved as {filename}.")
```

Step 4: Load and Explore the Data

```
# Load the dataset (for CSV file)
data = pd.read_csv(filename)
# Display first few rows
print("Dataset Preview:")
print(data.head())
```

Step 5: Preprocess the Data

```
# Separate features (X) and target (y)
X = data["X"].values.reshape(-1, 1)  # Feature matrix
y = data["y"].values  # Target vector
# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 6: Implement Bayesian Linear Regression Model

```
# Initialize the BayesianRidge model (which implements Bayesian Linear Regression)
bayesian_regressor = BayesianRidge()
# Fit the model on the training data
bayesian_regressor.fit(X_train, y_train)
```

 # Predict on the test data

y_pred = bayesian_regressor.predict(X_test)

Step 7: Visualize the Prior and Posterior Distributions

 # Plot the prior and posterior distributions of the parameters

fig, ax = plt.subplots(1, 2, figsize=(12, 6))

 # Plot prior distribution (assuming the model starts with a standard prior)

ax[0].set_title("Prior Distribution (Assumed)") ax[0].hist(np.random.normal(0, 1, 1000),

bins=50, alpha=0.7, color='blue', label="Prior") ax[0].legend()

 # Plot posterior distribution (after model fitting)

ax[1].set_title("Posterior Distribution (After Fitting)")
ax[1].hist(bayesian_regressor.coef_, bins=50, alpha=0.7, color='green',
label="Posterior") ax[1].legend()

plt.show()

Step 8: Evaluate the Model Performance #
Calculate the Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error (MSE):
{mse:.2f}")

Step 9: Visualize the Fit of the Model

 # Plot the true values and the predicted values

plt.figure(figsize=(8, 6))

plt.scatter(X_test, y_test, color="blue", label="True values")

plt.plot(X_test, y_pred, color="red", label="Predicted values",

linewidth=2)

plt.title("Bayesian Linear Regression: True vs Predicted Values")

plt.xlabel("X")

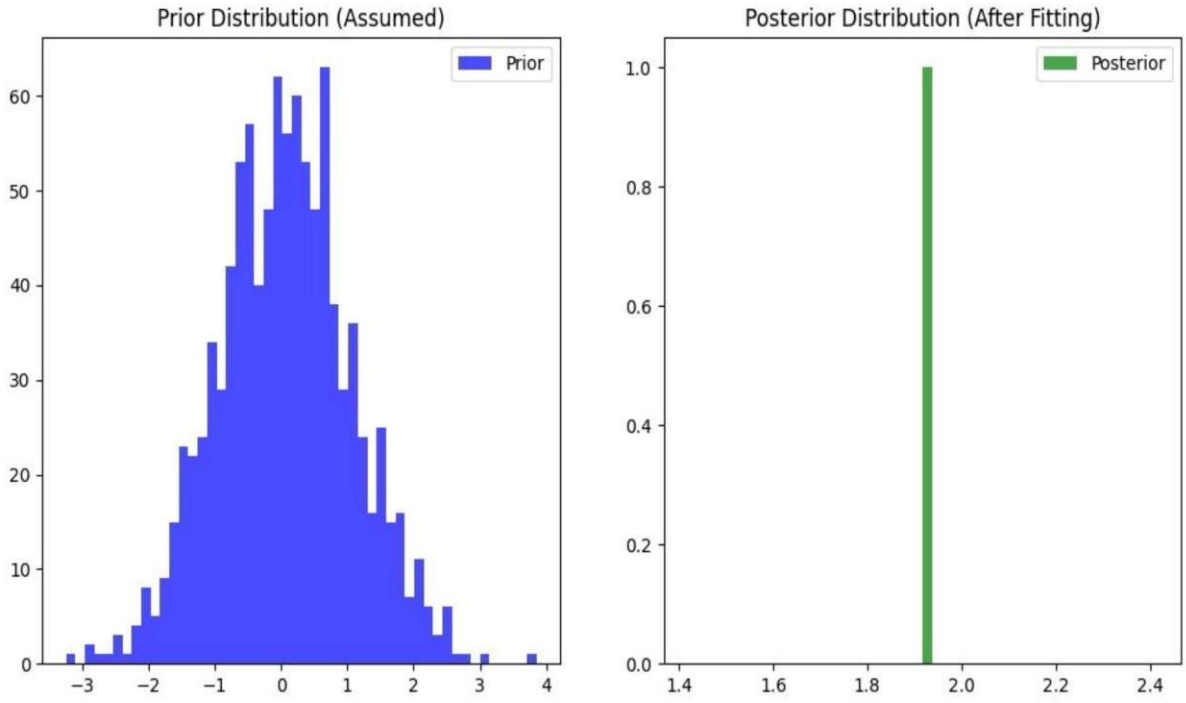
plt.ylabel("y")

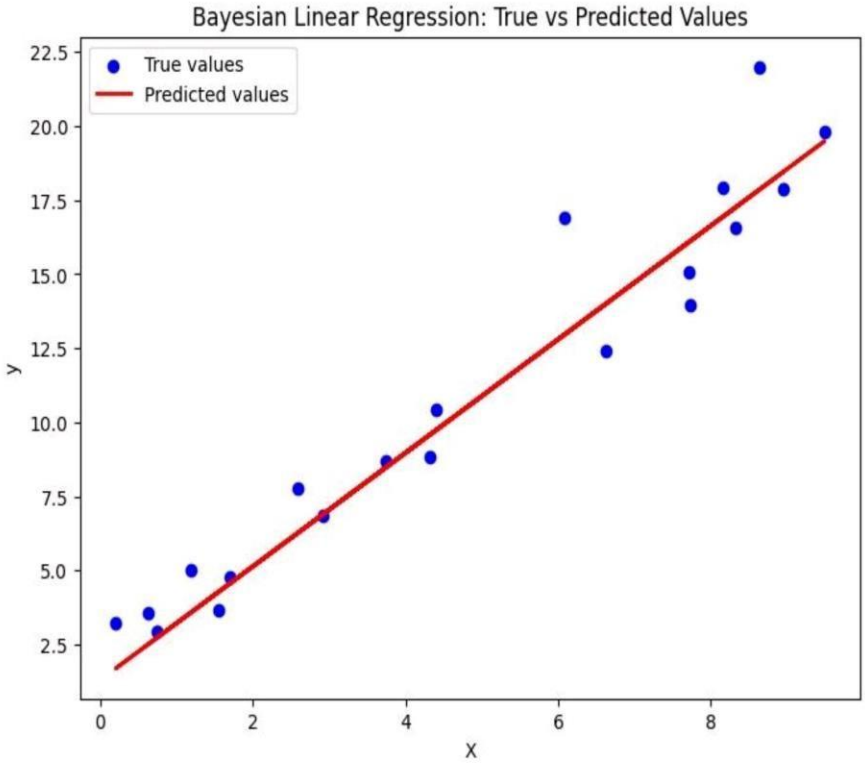
plt.legend()

plt.show()

Mean Squared Error (MSE): 3.9

Output :


Prior Distribution (Assumed)
Prior
Posterior Distribution (After Fitting)
Posterior


Bayesian Linear Regression: True vs Predicted Values
True values
Predicted values
y
X

6b. Implement Gaussian Mixture Models for density estimation and unsupervised clustering.

Code :

Step 1: Install Required Libraries

 # Install required libraries

```
!pip install matplotlib seaborn scikit-learn
```

Step 2: Import Required Libraries

 # Import necessary libraries

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.mixture import GaussianMixture
from sklearn.model_selection import train_test_split
from google.colab import files
```

Step 3: Create or Upload a Dataset

#Ask if the user has a CSV file to upload

```
print("Do you have a CSV file to upload? (yes/no)")
response = input().lower()

if response == "yes":
```

 # Upload the CSV file

```
uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:
```

 # Generate synthetic 2D data with two clusters for demonstration

```
  np.random.seed(42)
```

 # Generate data for two Gaussian distributions

```
  X1 = np.random.normal(loc=0, scale=1, size=(300, 2))  # Cluster 1: mean = 0, std = 1

  X2 = np.random.normal(loc=5, scale=1, size=(300, 2))  # Cluster 2: mean = 5, std = 1
```

 # Stack the data to create a dataset

```
  X = np.vstack([X1, X2])
```

 # Create DataFrame to simulate the CSV file for consistency
data = pd.DataFrame(X, columns=["Feature_1", "Feature_2"])
filename = "synthetic_data.csv"
data.to_csv(filename, index=False)
print(f"Synthetic dataset saved as {filename}.")

Step 4: Load and Explore the Dataset

 # Load the dataset (if CSV file is uploaded)
data = pd.read_csv(filename)
 # Display the first few rows
print("Dataset Preview:")
print(data.head())
 # Plot the data to visualize its structure
sns.scatterplot(data=data, x="Feature_1", y="Feature_2")
plt.title("Synthetic Data")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

Step 5: Fit a Gaussian Mixture Model (GMM)

 # Define the GMM model

n_components = 2 **# Number of Gaussian distributions (clusters)**

gmm = GaussianMixture(n_components=n_components, covariance_type='full',
random_state=42)

 # Fit the GMM model to the data

gmm.fit(data)

 # Predict the cluster labels for each data point

labels = gmm.predict(data)

 # Add the cluster labels to the dataset for visualization
data['Cluster'] = labels

 # Plot the clustered data

sns.scatterplot(data=data, x="Feature_1", y="Feature_2", hue="Cluster", palette="viridis",

marker="o")

plt.title("Gaussian Mixture Model Clustering")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

```
plt.legend()

plt.show()
```

Step 6: Visualize the Gaussian Mixture Model (GMM) Components

```
# Extract the means and covariances of the Gaussian components

means = gmm.means_

covariances = gmm.covariances_

# Plot the GMM components on top of the data

plt.figure(figsize=(8, 6))
# Plot data points

sns.scatterplot(data=data, x="Feature_1", y="Feature_2", hue="Cluster",
palette="viridis", marker="o", s=60, alpha=0.7)

# Plot the GMM ellipses for mean, covar in zip(means, covariances):

# Plot the Gaussian components as ellipses

v, w = np.linalg.eigh(covar)

v = 2.0 * np.sqrt(2.0) * np.sqrt(v)

# Scaling factor for the ellipse
u = w[0] / np.linalg.norm(w[0])

# Normalize the eigenvector

angle = np.arctan(u[1] / u[0])

# Create the ellipse

angle = angle * 180.0 / np.pi  # Convert to degrees
ellipse = plt.matplotlib.patches.Ellipse(mean, v[0], v[1], angle=angle, color='red', alpha=0.3)
plt.gca().add_patch(ellipse)


plt.title("GMM Clustering with Gaussian Components")
plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.legend()

plt.show()
```

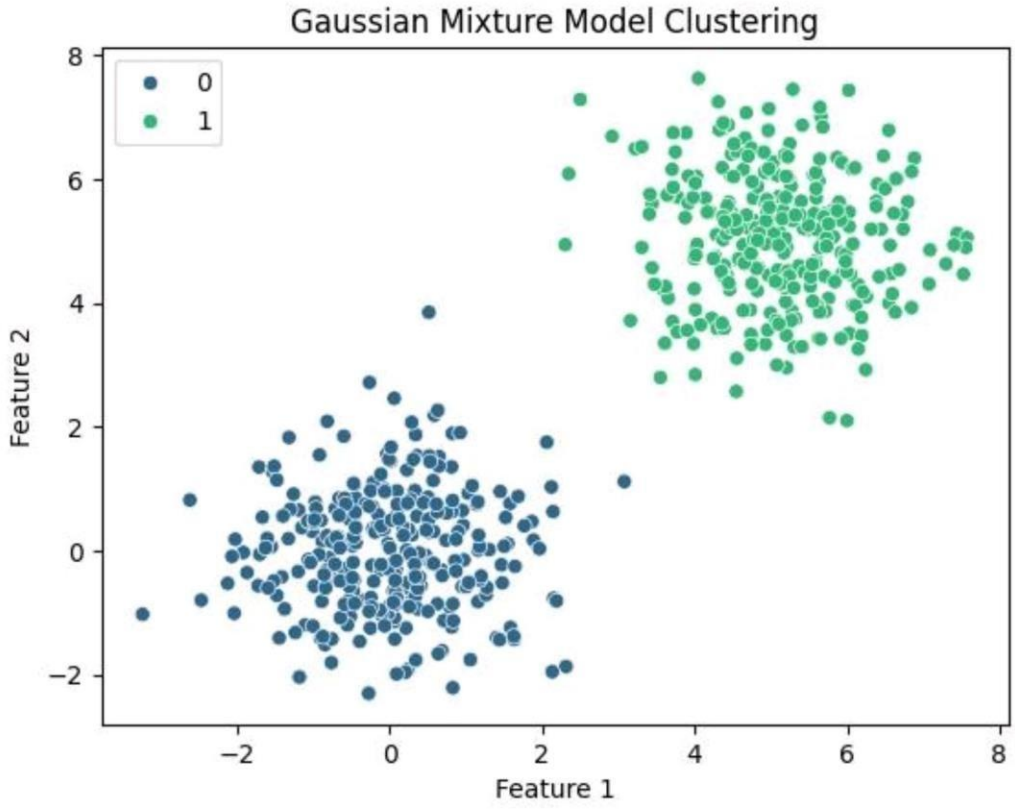
Step 7: Model Evaluation (Optional)

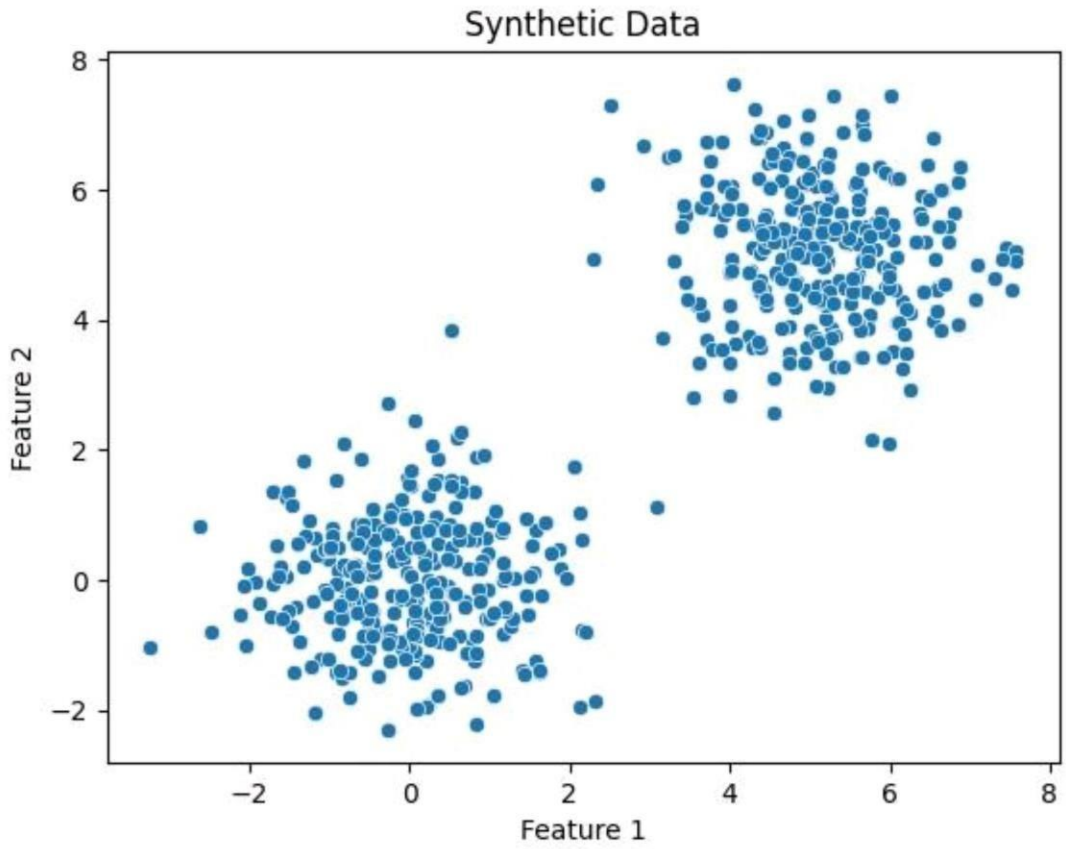
```
# Compute the log-likelihood of the data under the fitted GMM model
log_likelihood = gmm.score(data)
print(f"Log-Likelihood of the data: {log_likelihood:.2f}")
```

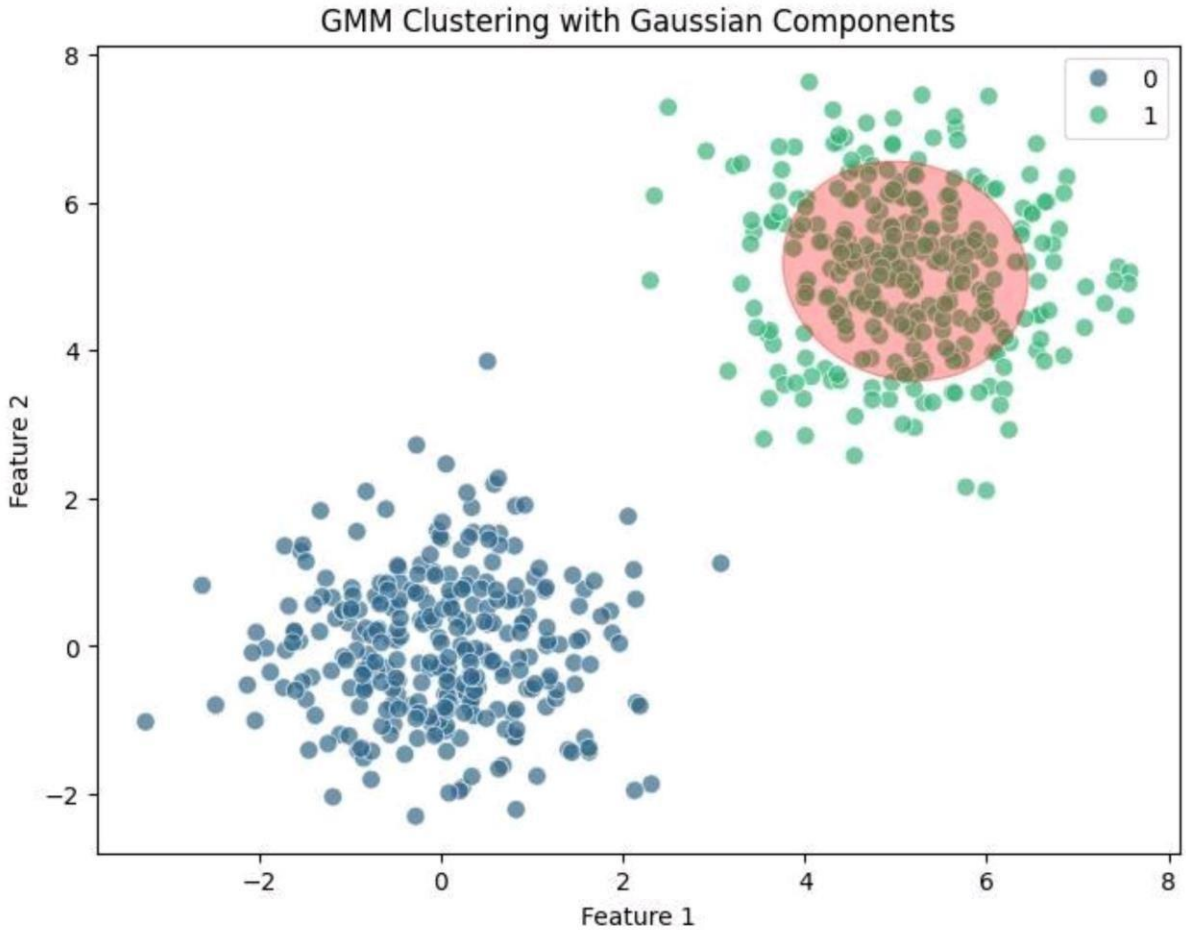
Step 8: Predict New Data Points

```
# Example of predicting the cluster for new data points
new_data = np.array([[1.5, 2.5], [4.5, 5.5], [7.0, 8.0]])

new_labels = gmm.predict(new_data)

# Print the predicted clusters for the new data
points

print("Predicted Clusters for New Data Points:")

for i, label in enumerate(new_labels):

    print(f"Data point {new_data[i]} is in Cluster {label}")
```

Output :




Synthetic Data
Feature 2
Feature 1
8
6
4
2
0
−2
−2
0
2
4
6
8


GMM Clustering with Gaussian Components
0
1
Feature 2
Feature 1
8
6
4
2
0
−2
−2
0
2
4
6
8

Practical 7 : Model Evaluation and Hyperparameter Tuning

7a. Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation

Code :

1. Import Necessary Libraries

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, KFold, StratifiedKFold, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Generate a Synthetic Dataset

```
# Create a synthetic dataset with 2 classes
X, y = make_classification(
    n_samples=1000, n_features=10, n_informative=8, n_redundant=2,
    n_clusters_per_class=1, random_state=42
)
# Convert to a DataFrame for visualization
df = pd.DataFrame(X, columns=[f'Feature_{i}' for i in range(1, 11)])
df['Target'] = y
# Display the first few rows
print(df.head())
```

3. Split Data into Train and Test Sets

```
# Split data into 80% training and 20% testing
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42,
stratify=y)
```

4. Define k-Fold Cross-Validation

```
kf = KFold(n_splits=5, shuffle=True, random_state=42)
print("k-Fold Cross-Validation:")
for train_index, val_index in kf.split(X_train):
    print("TRAIN:", train_index, "VALIDATION:", val_index)
```

5. Define Stratified k-Fold Cross-Validation

```
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
print("\nStratified k-Fold Cross-Validation:")
for train_index, val_index in skf.split(X_train, y_train):
    print("TRAIN:", train_index, "VALIDATION:", val_index)
```

6. Train and Evaluate Using k-Fold Cross-Validation

```
# Initialize model
model = RandomForestClassifier(random_state=42)
# Perform k-Fold Cross-Validation
accuracies = []
for train_index, val_index in kf.split(X_train):
    X_kf_train, X_kf_val = X_train[train_index], X_train[val_index]
    y_kf_train, y_kf_val = y_train[train_index], y_train[val_index]
    # Train model
    model.fit(X_kf_train, y_kf_train)
    # Validate model
    y_pred = model.predict(X_kf_val)
    accuracy = accuracy_score(y_kf_val, y_pred)
    accuracies.append(accuracy)
print(f"Average Accuracy from k-Fold: {np.mean(accuracies):.2f}")
```

7. Hyperparameter Tuning Using GridSearchCV

```
# Define parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
}
# Perform GridSearchCV with Stratified k-Fold
grid_search = GridSearchCV(
    estimator=RandomForestClassifier(random_state=42),
    param_grid=param_grid,
    cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
    scoring='accuracy',
    n_jobs=-1,
    verbose=1
)
# Fit to training data
grid_search.fit(X_train, y_train)
print("Best Parameters:", grid_search.best_params_)
print("Best Cross-Validation Accuracy:", grid_search.best_score_)
```

8. Evaluate the Final Model

```
# Use the best model for evaluation
best_model = grid_search.best_estimator_
# Predict on test data
y_test_pred = best_model.predict(X_test)
# Evaluate performance
print("\nTest Accuracy:", accuracy_score(y_test, y_test_pred))
print("\nClassification Report:\n", classification_report(y_test, y_test_pred))
```

 # Confusion matrix

conf_matrix = confusion_matrix(y_test, y_test_pred)

sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class 1'], yticklabels=['Class 0', 'Class 1'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

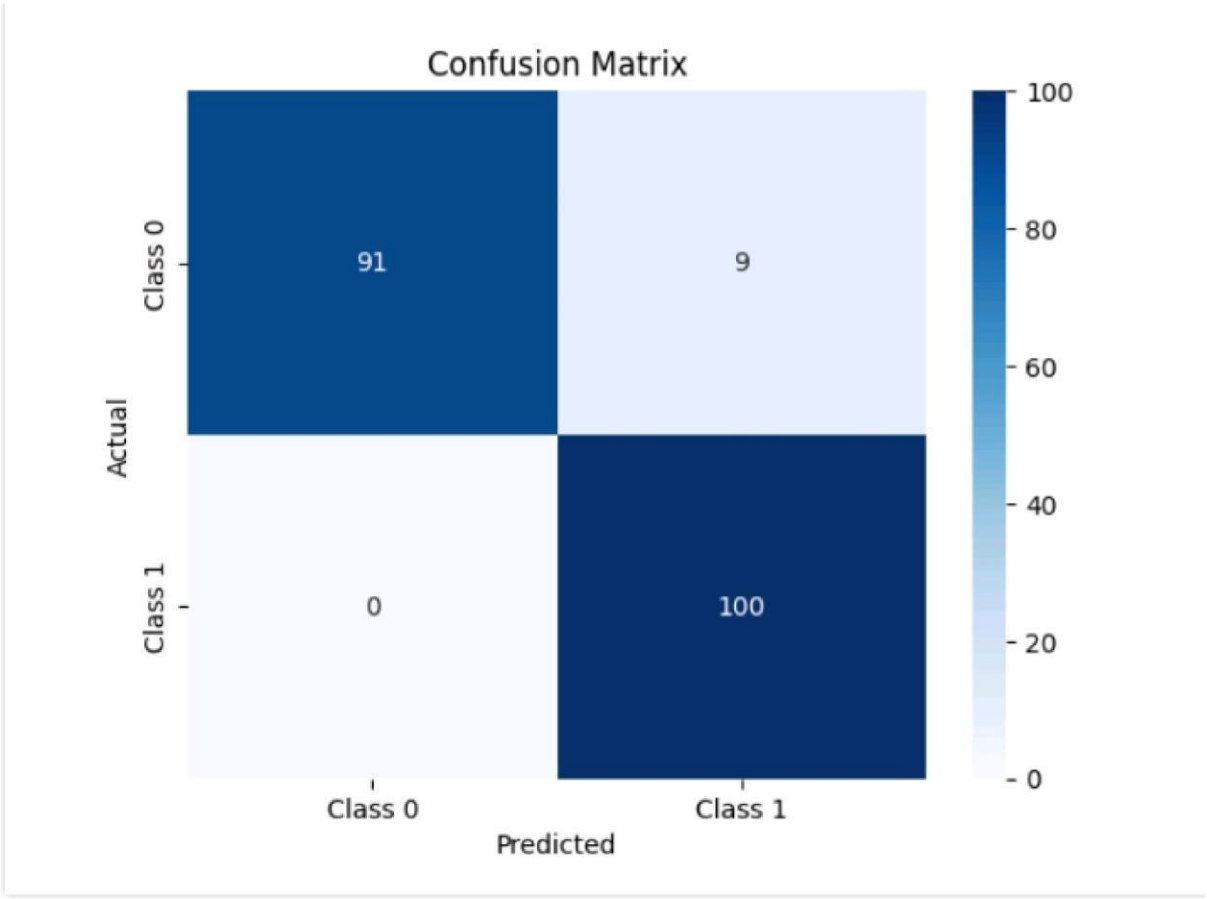
plt.show()

Output :

```
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  Feature_6  \
0  -3.358483   3.159918   0.827163   0.069638  -6.715639  -2.708559
1   2.071819  -4.055419  -2.615940  -2.599432   3.053752   0.366795
2  -0.633460   0.712482   2.024390  -0.432639  -1.307929   0.419320
3  -0.464478   0.892442   2.521010   2.766580  -1.933734  -1.418018
4   1.042426  -1.192605  -2.071386  -0.131231   0.545377   0.379060

   Feature_7  Feature_8  Feature_9  Feature_10  Target
0   0.183206   1.113502   1.730759    1.228394       1
1  -0.392171  -1.191720  -1.220516    1.899925       0
2  -1.469510  -0.719051   1.155005    2.018026       0
3   1.391760  -2.430279   1.308295   -0.270896       1
4  -0.062978  -1.325591   2.037936    0.115414       0
k-Fold Cross-Validation:
TRAIN: [  0   1   3   4   5   6   8   9  11  12  13  14  15  16  17  18  19  20
  21  22  24  25  26  27  28  32  34  35  36  37  38  40  41  42  43  44
  45  46  47  48  50  51  52  53  55  56  57  58  59  60  61  62  64  68
  69  70  71  73  74  75  79  80  82  83  85  87  88  89  90  91  92  93
  94  95  98  99 100 102 103 104 105 106 107 108 111 112 113 114 115 116
 117 119 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
 138 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 156 157
 158 159 160 161 162 163 164 165 166 167 169 170 171 172 173 175 176 177
 178 179 180 181 182 183 184 185 186 187 188 189 190 191 193 194 195 196
 197 200 201 202 203 205 206 207 212 213 214 216 217 219 220 221 222 223
 224 225 226 227 228 229 230 232 233 234 236 237 238 239 240 241 242 243
 245 246 247 248 249 251 252 253 255 256 257 258 259 261 262 263 264 267
 268 269 270 271 272 273 274 276 277 278 279 280 282 283 284 285 287 288
 289 290 291 292 293 295 297 298 299 300 301 303 304 305 307 308 309 310
 311 312 313 315 317 318 319 320 321 322 324 325 328 329 330 331 332 334
```

```
 674 675 676 677 678 679 680 681 682 683 684 685 686 687 689 690 691 693
 694 695 697 698 699 700 701 702 703 704 706 707 708 709 710 711 712 713
 714 716 717 718 722 724 725 726 727 728 729 730 731 732 733 734 735 736
 737 739 740 742 743 745 747 748 749 751 752 753 755 756 757 759 761 762
 763 765 766 767 768 769 770 771 772 773 774 775 778 779 780 781 782 784
 785 788 789 790 791 792 793 794 797 799] VALIDATION: [  2   7  10  23  29  30  31  33  39  49  54  63  65  66  67  72  76  77
  78  81  84  86  96  97 101 109 110 118 120 137 139 155 168 174 192 198
 199 204 208 209 210 211 215 218 231 235 244 250 254 260 265 266 275 281
 286 294 296 302 306 314 316 323 326 327 333 336 346 352 357 360 361 365
 367 368 377 383 388 393 395 398 409 422 423 425 428 432 446 456 464 481
 483 486 490 506 512 513 515 519 521 525 526 529 532 533 534 537 545 568
 570 578 589 591 594 595 596 604 608 610 621 622 628 635 637 640 641 644
 652 655 656 658 662 666 667 670 688 692 696 705 715 719 720 721 723 738
 741 744 746 750 754 758 760 764 776 777 783 786 787 795 796 798]
TRAIN: [  0   1   2   3   4   5   7   8   9  10  12  13  14  15  16  17  19  20
  21  22  23  25  26  27  29  30  31  32  33  34  35  36  37  38  39  40
  45  46  47  48  49  50  52  53  54  57  58  59  62  63  64  65  66  67
  68  71  72  75  76  77  78  80  81  84  85  86  87  88  91  93  94  95
  96  97  98  99 100 101 102 103 105 106 107 109 110 111 112 113 115 116
 117 118 119 120 121 122 123 124 125 126 127 128 129 130 134 137 138 139
 141 142 143 144 146 147 149 150 151 152 153 154 155 156 157 159 160 161
 162 166 168 169 170 171 172 173 174 175 179 180 183 184 185 186 187 188
 189 190 191 192 194 195 197 198 199 200 201 202 203 204 205 206 207 208
 209 210 211 214 215 216 217 218 219 221 222 224 225 226 228 229 230 231
 232 233 235 236 237 238 240 241 242 243 244 245 246 249 250 251 252 253
 254 255 256 257 258 260 261 262 263 265 266 267 268 269 270 271 272 273
 274 275 276 277 278 279 280 281 282 283 284 286 287 288 289 293 294 295
 296 297 298 301 302 303 304 305 306 307 308 310 312 313 314 315 316 317
 318 320 321 322 323 324 325 326 327 330 333 335 336 337 338 339 340 341
```

```
 450 455 459 460 463 470 472 475 480 481 483 494 496 502 503 505 514 519
 521 523 529 533 553 555 560 563 565 579 585 590 594 600 603 620 630 631
 634 635 637 638 644 646 648 649 651 673 675 686 691 693 708 709 715 720
 729 730 738 747 753 759 767 771 774 776 777 780 782 783 784 796]
Average Accuracy from k-Fold: 0.95
Fitting 5 folds for each of 36 candidates, totalling 180 fits
Best Parameters: {'max_depth': None, 'min_samples_split': 5, 'n_estimators': 50}
Best Cross-Validation Accuracy: 0.96

Test Accuracy: 0.955

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.91      0.95       100
           1       0.92      1.00      0.96       100

    accuracy                           0.95       200
   macro avg       0.96      0.96      0.95       200
weighted avg       0.96      0.95      0.95       200
```


Confusion Matrix
Class 0
Class 1
Actual
91
9
0
100
Class 0
Class 1
Predicted
100
80
60
40
20
0

7b. Systematically explore combinations of hyperparameters to optimize model performance.(use grid and randomized search)

Code :

1. Import Necessary Libraries

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Generate a Synthetic Dataset

```
# Generate a binary classification dataset
X, y = make_classification(
    n_samples=1000, n_features=12, n_informative=8, n_redundant=2,
    n_clusters_per_class=1, flip_y=0.03, random_state=42
)
# Convert to a DataFrame for visualization
df = pd.DataFrame(X, columns=[f'Feature_{i}' for i in range(1, 13)])
df['Target'] = y
# Display the first few rows
print(df.head())
```

3. Split Data into Train and Test Sets

```
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

4. Define the Model

 # Initialize a Random Forest classifier

```
model = RandomForestClassifier(random_state=42)
```

5. Hyperparameter Tuning Using Grid Search

```
# Define a parameter grid for Grid Search
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

 # GridSearchCV with 5-fold cross-validation

```
grid_search = GridSearchCV(
    estimator=model,
    param_grid=param_grid,
    scoring='accuracy',
    cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
    verbose=1,
    n_jobs=-1
)
```

 # Fit the model

```
grid_search.fit(X_train, y_train)
```

 # Best parameters and score from Grid Search

```
print("Best Parameters from Grid Search:", grid_search.best_params_)
print("Best Cross-Validation Accuracy from Grid Search:", grid_search.best_score_)
```

6. Hyperparameter Tuning Using Randomized Search

```
from scipy.stats import randint
```

 # Define a parameter distribution for Randomized Search

```
param_dist = {
    'n_estimators': randint(50, 300),
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': randint(2, 15),
    'min_samples_leaf': randint(1, 10)
}
```

 # RandomizedSearchCV with 5-fold cross-validation

```
random_search = RandomizedSearchCV(
    estimator=model,
    param_distributions=param_dist,
    n_iter=50,  # Number of random combinations to try
    scoring='accuracy',
    cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
    verbose=1,
    n_jobs=-1,
    random_state=42
)
```

 # Fit the model

```
random_search.fit(X_train, y_train)
```

 # Best parameters and score from Randomized Search

```
print("Best Parameters from Randomized Search:", random_search.best_params_)
print("Best Cross-Validation Accuracy from Randomized Search:",
random_search.best_score_)
```

7. Evaluate the Best Model

```
# Select the best model from Grid Search and Randomized Search
best_model = random_search.best_estimator_  # Or use grid_search.best_estimator_
```

 # Predict on test data

```
y_test_pred = best_model.predict(X_test)
```

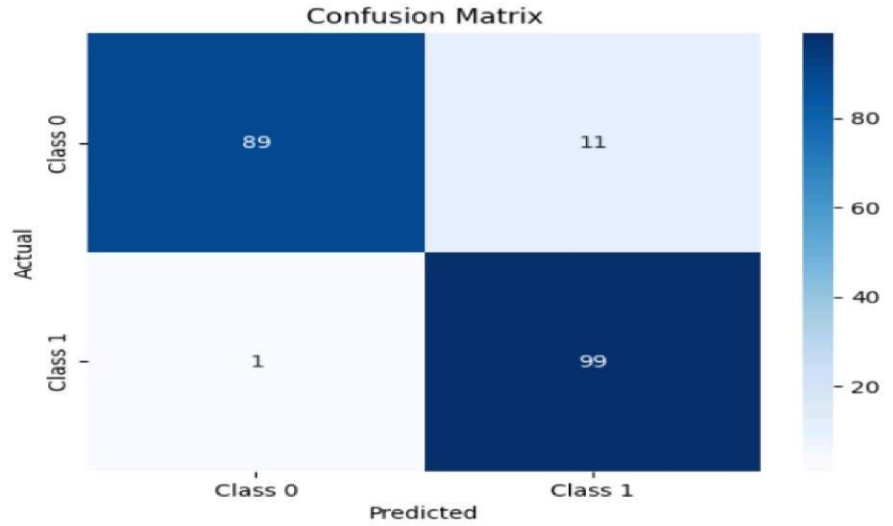
 # Evaluate the performance

```
print("\nTest Accuracy:", accuracy_score(y_test, y_test_pred))

print("\nClassification Report:\n", classification_report(y_test, y_test_pred))

# Confusion Matrix

conf_matrix = confusion_matrix(y_test, y_test_pred)

sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class 1'], yticklabels=['Class 0', 'Class 1'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()
```

Output :

```
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  Feature_6  \
0   0.013650   0.607473  -2.096916   2.867232   2.504360   0.784101
1   0.107199   0.105735  -3.843343   1.524052  -1.619824   0.778334
2  -1.779086  -5.219831  -0.738488   2.108084  -0.803833  -3.431122
3  -4.310656  -2.268569   1.864943  -1.246116   1.260794  -2.007664
4  -3.195179  -0.671327   3.720485   0.356661   0.819486   2.670230

   Feature_7  Feature_8  Feature_9  Feature_10  Feature_11  Feature_12  Target
0  -0.497744  -0.482072   1.112773    1.641637   -2.689832   -0.480311       1
1   0.551177  -1.843583  -0.110132   -0.494739   -0.985276   -0.978400       0
2   1.346120  -0.858351  -0.792415   -2.260815    0.238780    3.029952       0
3  -0.824133  -2.277449   0.936206    1.255903    1.386278   -0.321200       0
4   1.857477  -3.410944  -1.773719    0.656476    3.534189   -1.704889       0
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Best Parameters from Grid Search: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Best Cross-Validation Accuracy from Grid Search: 0.9487499999999999
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Parameters from Randomized Search: {'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 9, 'n_estimators': 285}
Best Cross-Validation Accuracy from Randomized Search: 0.9487499999999999

Test Accuracy: 0.94
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.89      0.94       100
           1       0.90      0.99      0.94       100

    accuracy                           0.94       200
   macro avg       0.94      0.94      0.94       200
weighted avg       0.94      0.94      0.94       200
```



In Bayesian Learning, the model predicts based on the probabilities:

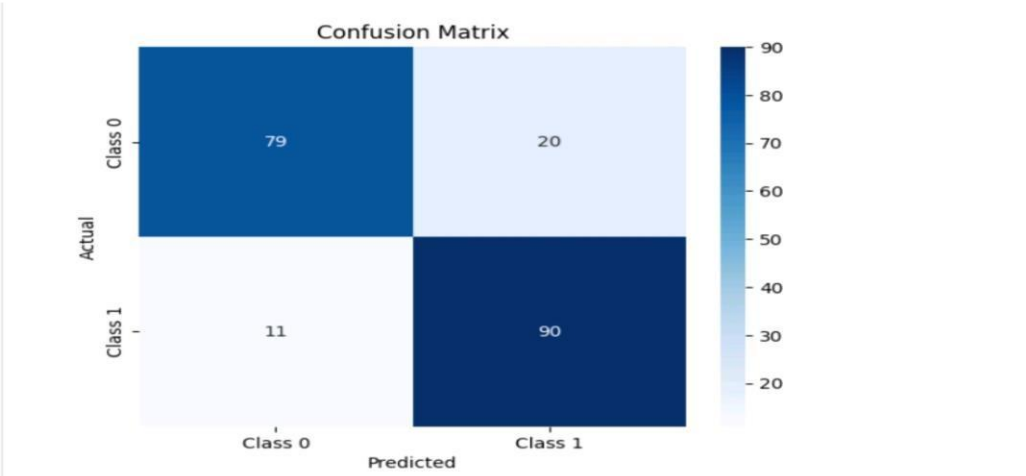
- **Prior Probability (P(C)P(C)P(C)):** The likelihood of each class based on historical data.
- **Likelihood (P(X|C)P(X|C)P(X|C)):** The probability of the data given a class.
- **Posterior Probability (P(C|X)P(C|X)P(C|X)):** Calculated using Bayes' theorem: P(C|X)=P(X|C)·P(C)P(X)P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}P(C|X)=P(X)P(X|C)·P(C)

 # Example: Compute posterior probabilities for the first test sample

```
sample = X_test[0].reshape(1, -1)

posterior_probs = model.predict_proba(sample)

print(f"Sample Features: {sample}")

print(f"Posterior Probabilities: {posterior_probs}")

print(f"Predicted Class: {model.predict(sample)}")
```

Output :

```
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  Feature_6  \
0  -1.732538   5.260112  -2.952194  -4.603768   2.235848   1.928893
1   2.072914   2.240572  -1.385104  -2.514962  -0.984756   1.436260
2  -0.263106   1.527781  -1.872414  -0.028009   1.612809   3.264194
3  -0.164349  -0.550131  -0.019503  -0.765000   2.273523   2.084217
4  -1.419423   1.015324  -0.864441  -0.009297   0.385404   0.449093

   Feature_7  Feature_8  Target
0  -0.101845   3.193487       0
1  -1.255271   2.089872       0
2  -1.296421   1.537870       0
3  -0.321931   0.426253       0
4  -0.029007  -1.902917       1
Test Accuracy: 0.84

Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.80      0.84        99
           1       0.82      0.89      0.85       101

    accuracy                           0.84       200
   macro avg       0.85      0.84      0.84       200
weighted avg       0.85      0.84      0.84       200
```




```
Sample Features: [[ 1.94397502  0.33753289 -0.77968371  2.63667244 -1.62323864  1.87859635
  -2.20522563  2.09632056]]
Posterior Probabilities: [[0.07067404 0.92932596]]
Predicted Class: [1]
```

M.Sc. IT. SEM 3 Server Virtualization on VMWare platform

Sr. No.	Practical No	Index	Page No	Date	Sign
1	1a	Configure and use vCenter Server Appliance			
2	2a	Adding and Configuring vSphere Standard Switch			
3	2b	Configure Access to an iSCSI datastore			
4	3a	Create and manage VMFS datastore			
5	3b	Configure Access to an NFS datastore			
6	3c	Deploy a new virtual machine from a template and clone a virtual machine			
7	4a	Create a content library to clone and deploy virtual machines			
8	5	Use vSphere vMotion and vSphere Storage vMotion to migrate virtual machines			
9	7b	Use the system monitoring tools to reflect the CPU workload			
10	8	Use the vCenter Server Appliance alarm feature			
11	9	Use vSphere HA functionality			
12	10a	Implement a vSphere DRS cluster			

Practical 1: Configure and use vCenter Server Appliance.

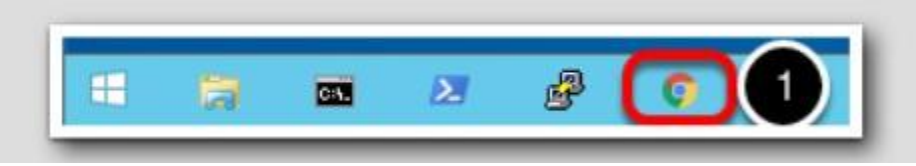
ESXi Host Client: The VMware Host Client is an HTML5-based client that is used to connect to and manage single ESXi hosts. You can use the VMware Host Client to perform administrative and basic troubleshooting tasks, as well as advanced administrative tasks on your target ESXi host. You can also use the VMware Host Client to conduct emergency management when vCenter Server is not available.

It is important to know that the VMware Host Client is different from the vSphere Web Client, regardless of their similar user interfaces. You use the vSphere Web Client to connect to vCenter Server and manage multiple ESXi hosts, whereas you use the VMware Host Client to manage a single ESXi host.

This lesson will walk through some of the most frequently used features in the ESXi Host Client.

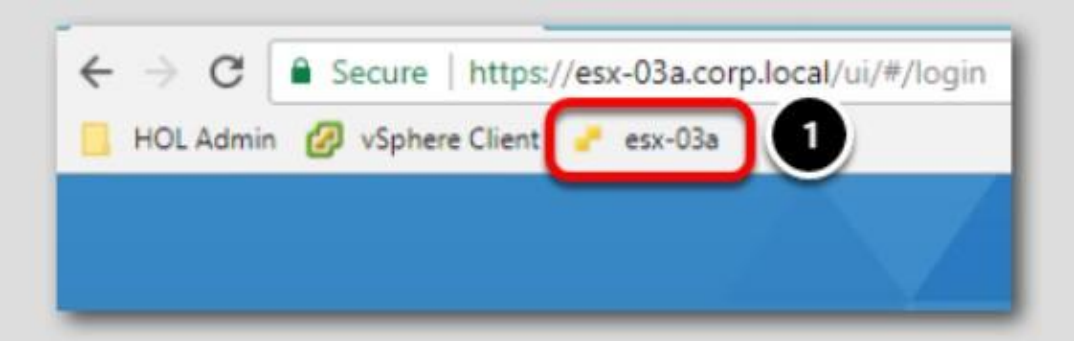
Launch Chrome

1.Click on the Chrome Icon on the Windows Quick Launch Task Bar.



Select esx-03a

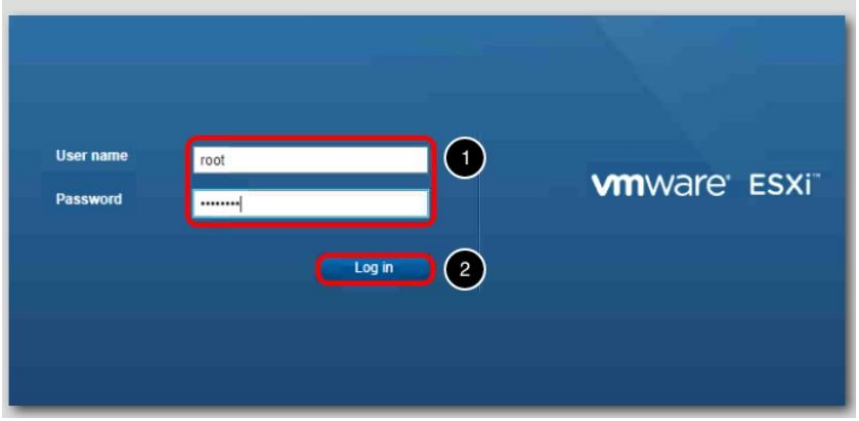
1.From the Bookmarks bar, select esx-03a



Login

Login with the following credentials:

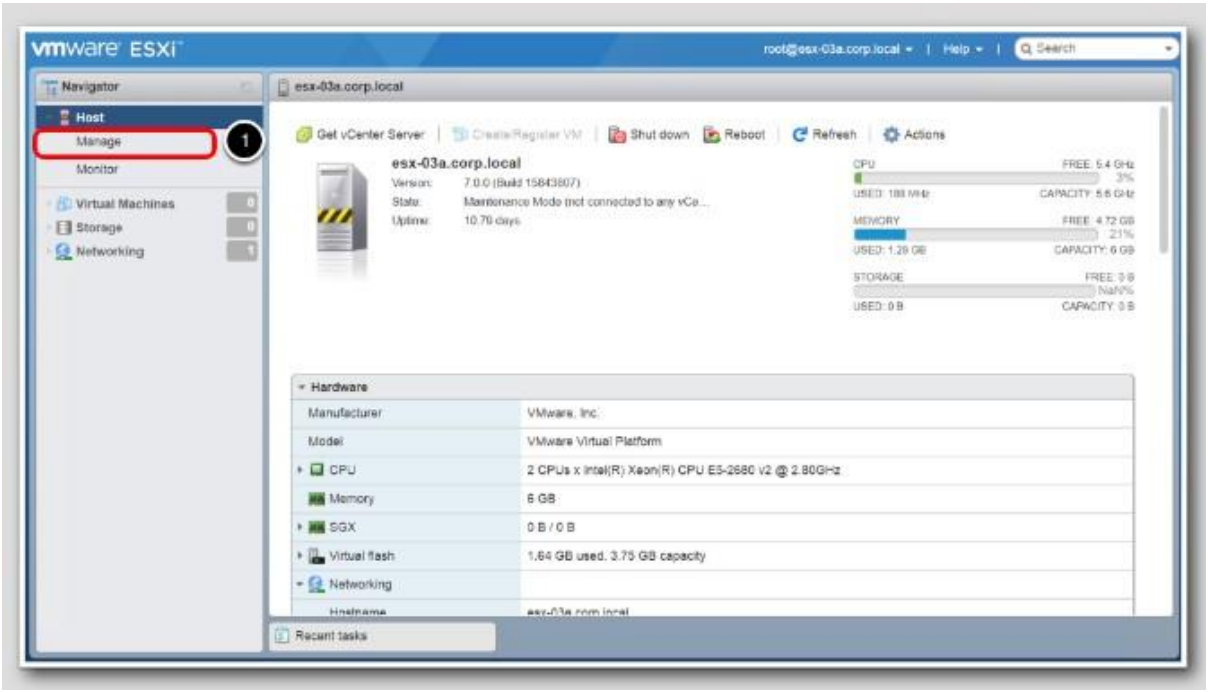
1. User name: root
2. Password: VMware1!



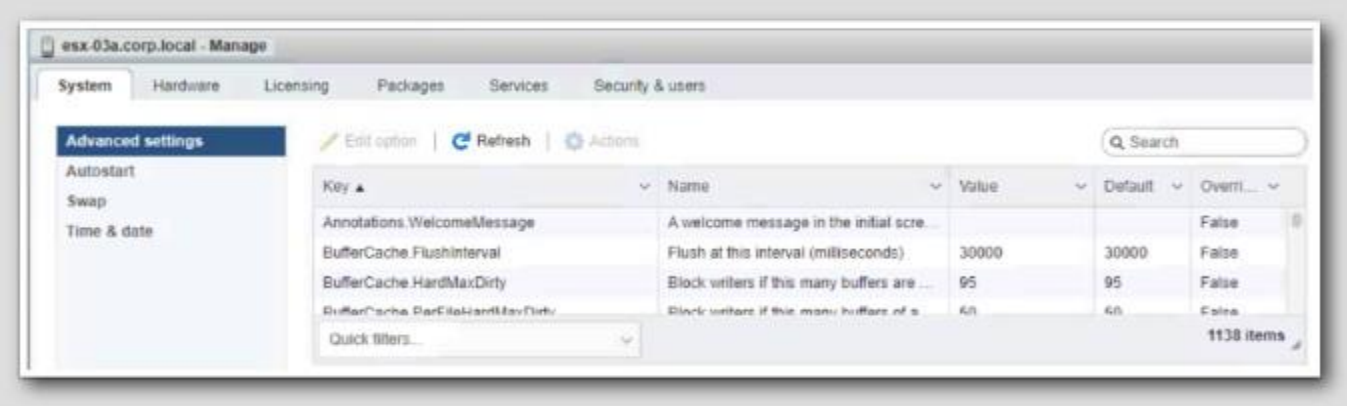
Click the Log in button. The ESXi Host, in this case, esx-03aesx-03a, can now be directly managed. This can be useful in test/dev environments where a vCenter Server is not present or in a production environment where the vCenter Server is not reachable.

ESXi Host Client

1. Click on Manage.

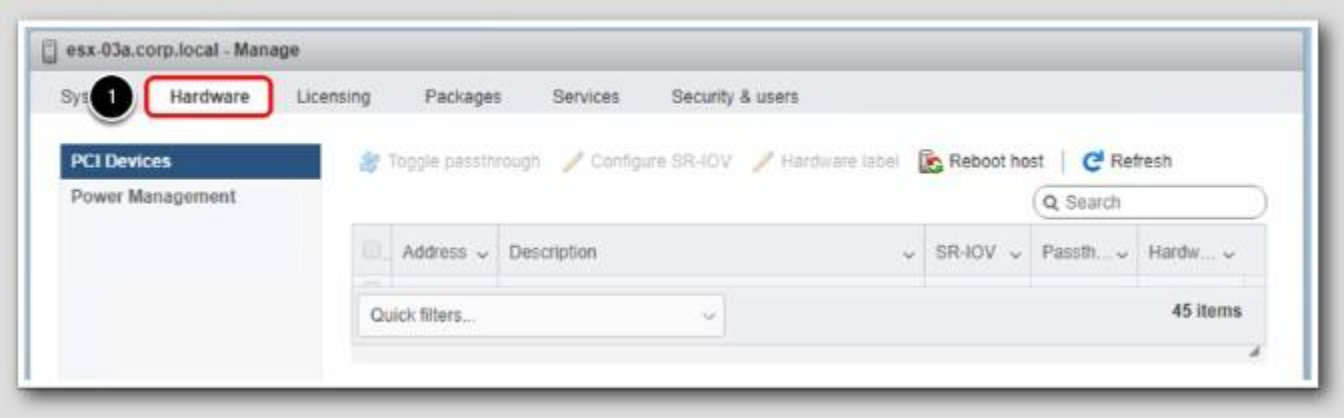


2. On the System tab, the most common options set here are the date and time for the host. It can be set and synchronized with an NTP server or set manually. In addition, Autostart settings for the host can be configured here as well.

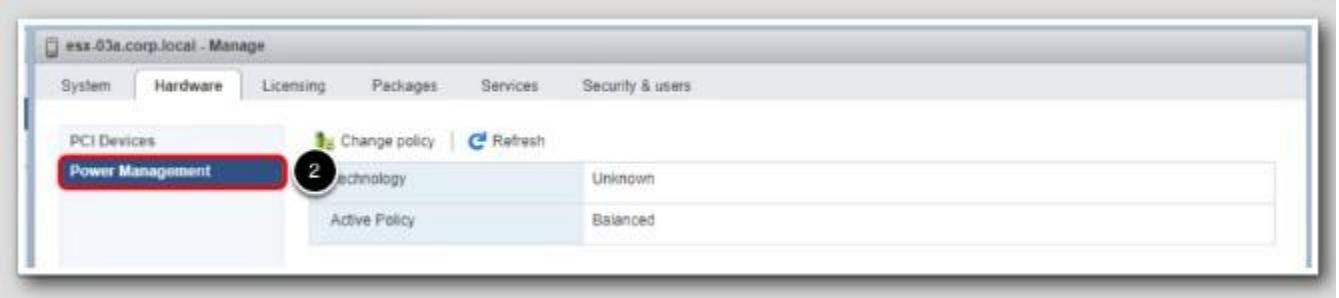


Hardware

1. Click on the Hardware tab.



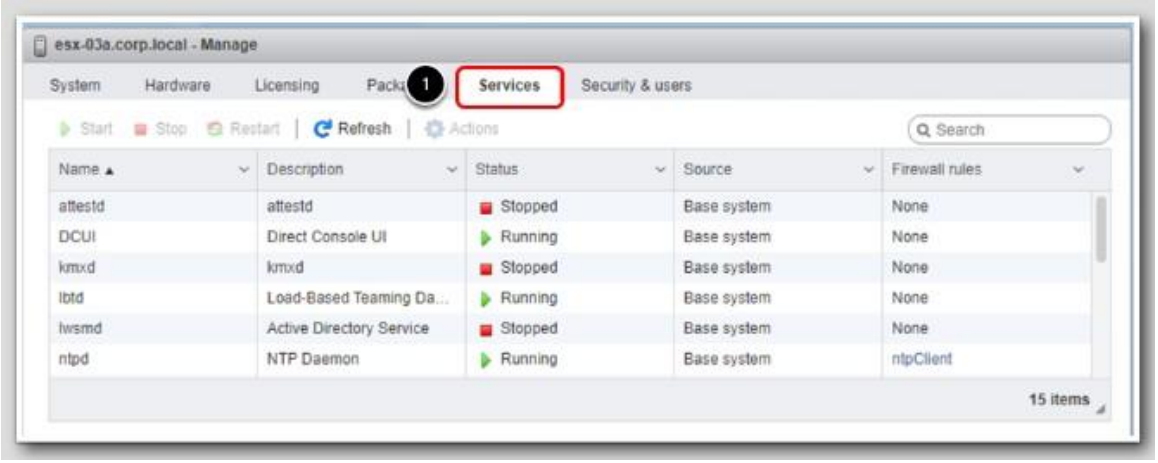
2. Click Power Management. This is where power management policies can be set for the host.



Services

1.Click the Services tab

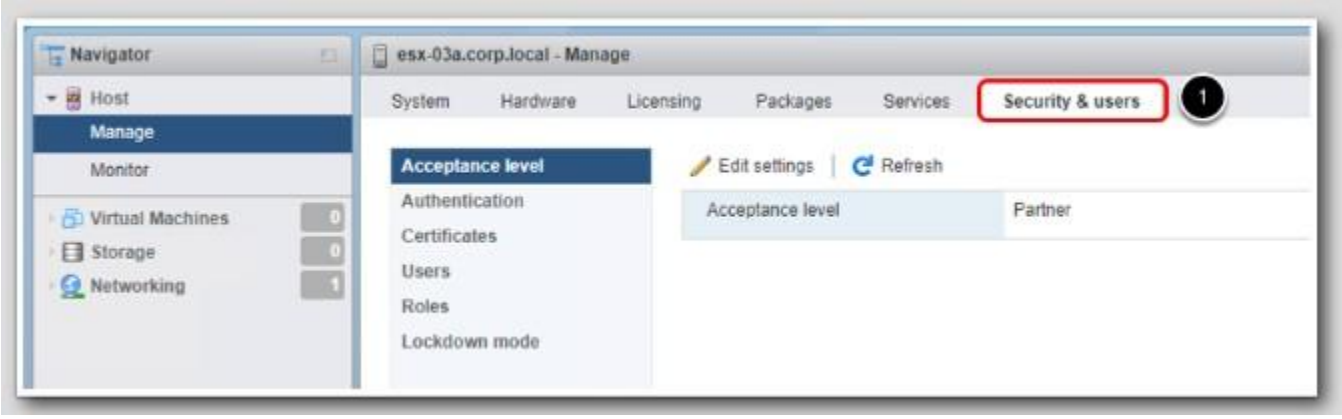
Services like SSH access and the Direct Console UI can be stopped and started from this screen



Security and Users

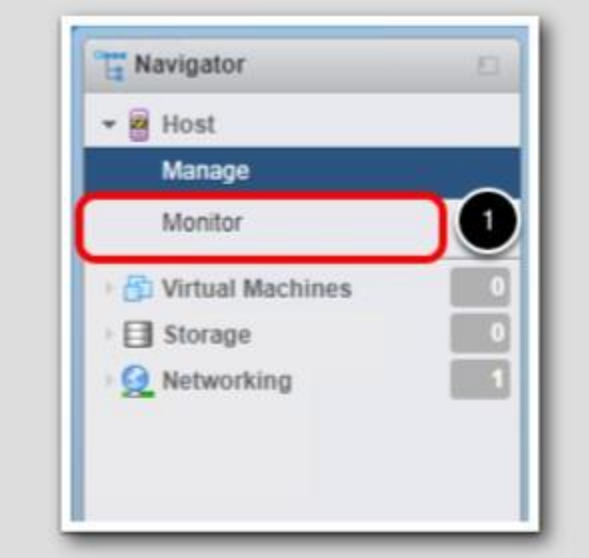
1.Click on Security & user.

On the Security & Users tab, security options such as authentication to Active Directory and Certificates can be set here. There is also the ability to create additional roles and user accounts for the host itself. This option uses accounts that are local only to the host and not shared with any other hosts or vCenter Server. vCenter Server is set up to use single sign-on which makes account management much easier.



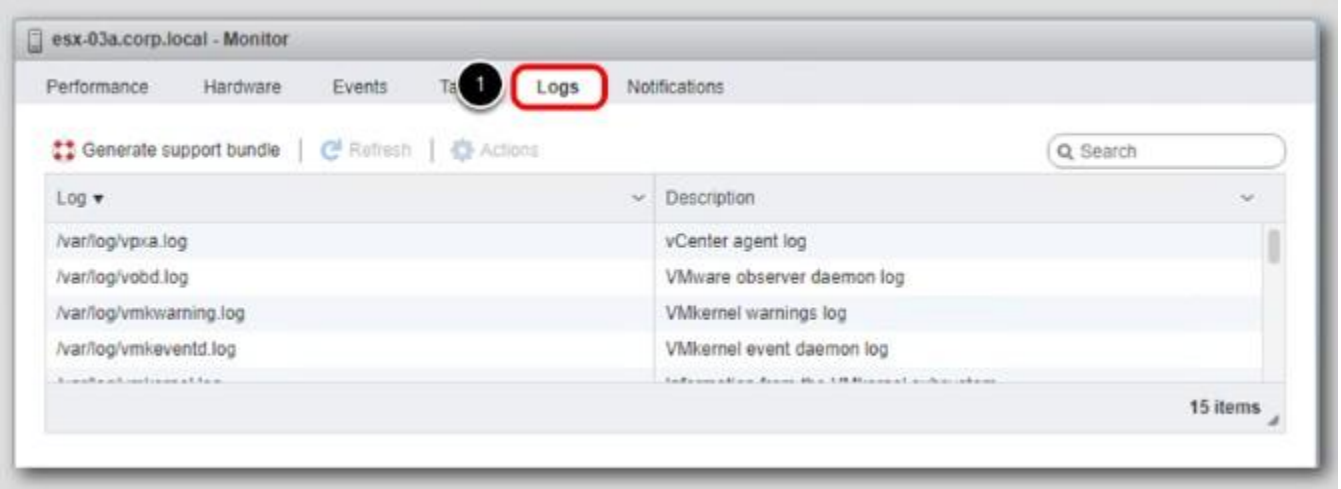
Monitor

1. Click on Monitor. The Monitor section includes Performance Charts, Hardware monitoring, an event log and other useful monitoring information.



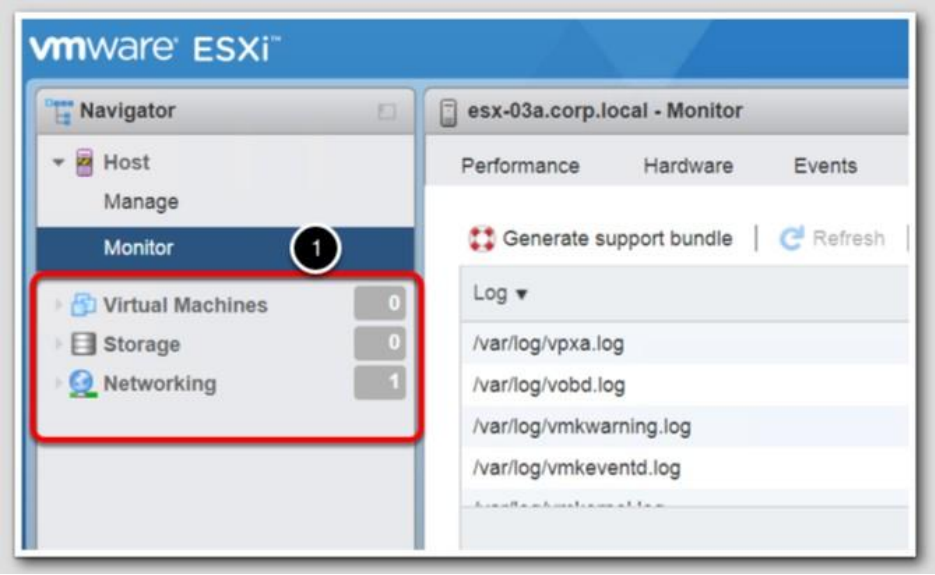
Logs

1. Click the Logs tab. On the Logs tab, a support bundle can be created that includes log files and system information that can be helpful in troubleshooting issues.



VMs, Storage and Networking

1. In addition to managing and monitoring the host, Virtual Machines can be created, Storage and Networking can be configured at the host level.


vmware ESXi
Navigator
Host
Manage
Monitor
1
Virtual Machines
0
Storage
0
Networking
1
esx-03a.corp.local - Monitor
Performance
Hardware
Events
Generate support bundle
Refresh
Log
/var/log/vpxa.log
/var/log/vobd.log
/var/log/vmkwarning.log
/var/log/vmkeventd.log

Practical 2: Adding and Configuring vSphere Standard Switch

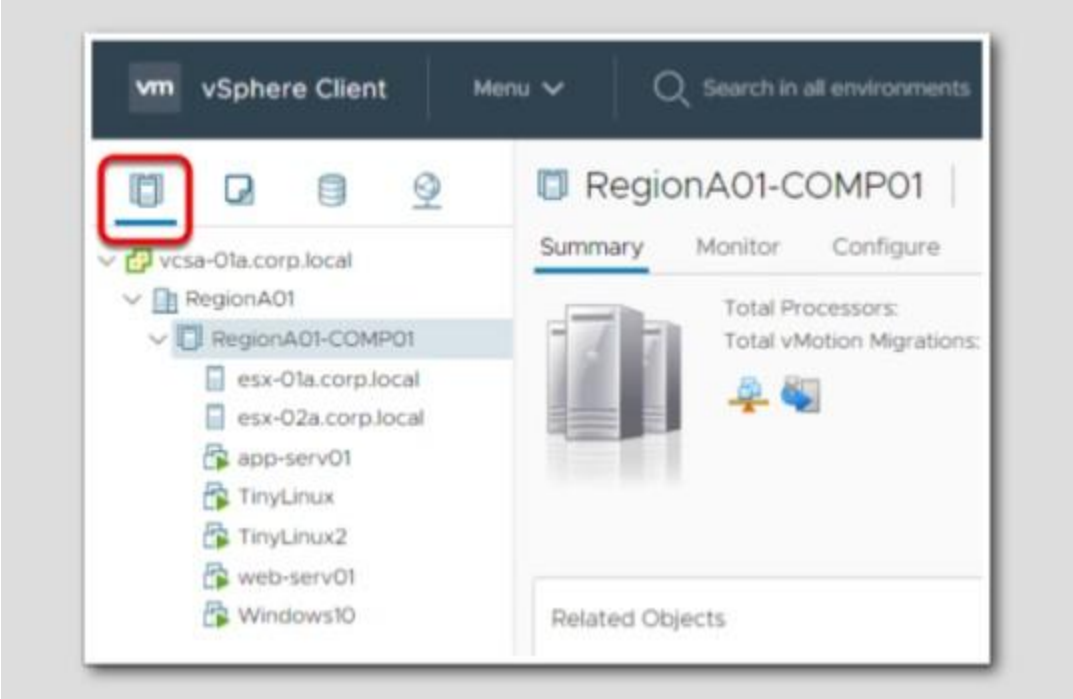
The following lesson will walk you through the process of creating and configuring the vSphere Standard Switch.

Adding a Virtual Machine Port Group with the vSphere Client

If you are not already logged in, launch the Chrome browser from the desktop and log in to the vSphere Web Client.

1. Click the "Use Windows session authentication" check box
2. Click "Login"

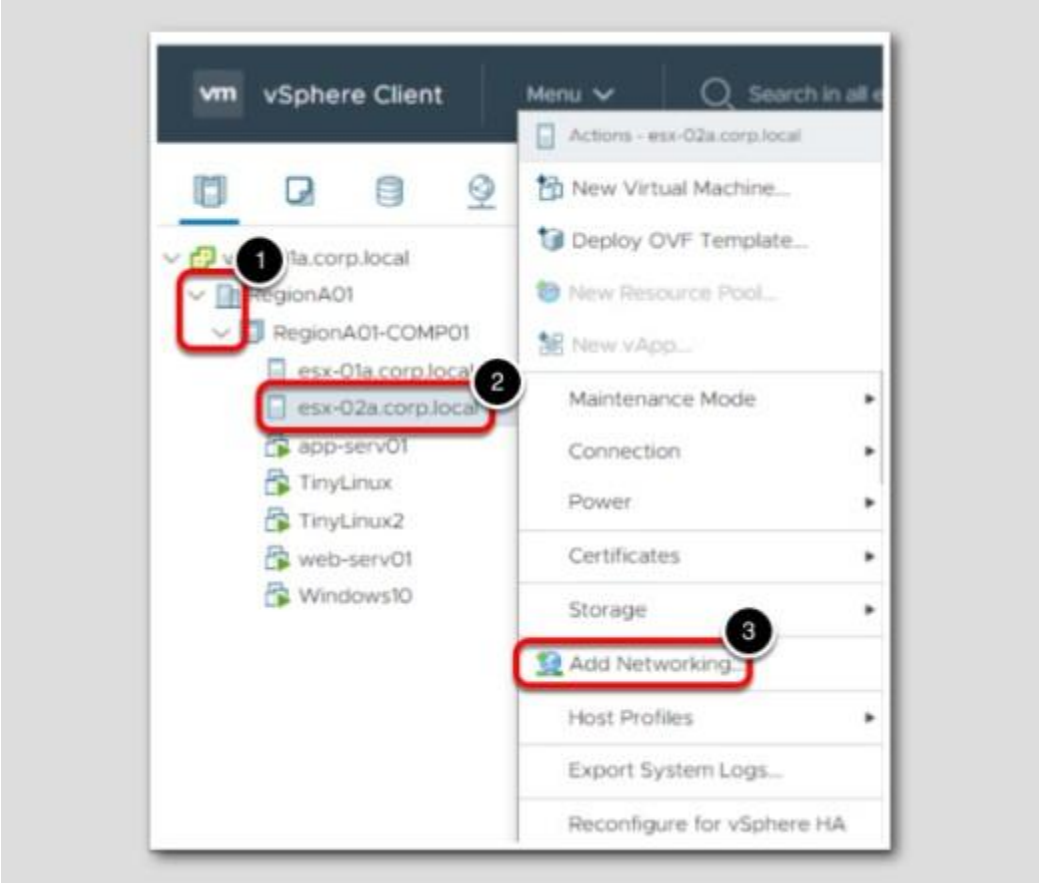
Select Hosts and Clusters



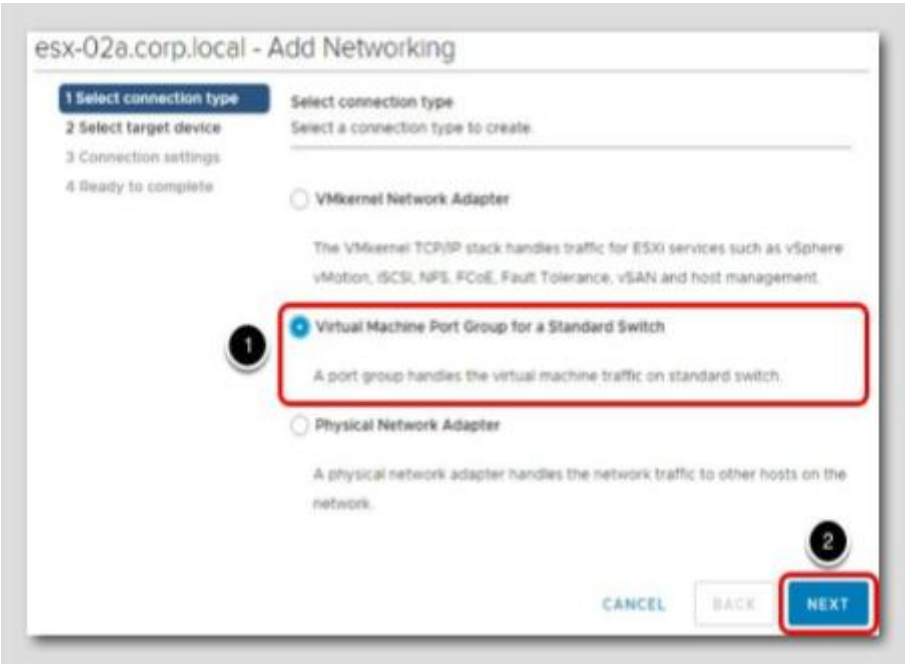
If you are not directed to "Hosts and Clusters", click the icon for it.

Add Networking

1. Under vcsa-01a.corp.local, expand RegionA01 and then RegionA01-COMP01.
2. Next, right-click on esx-02a.corp.local in the Navigator.
3. Select Add Networking....



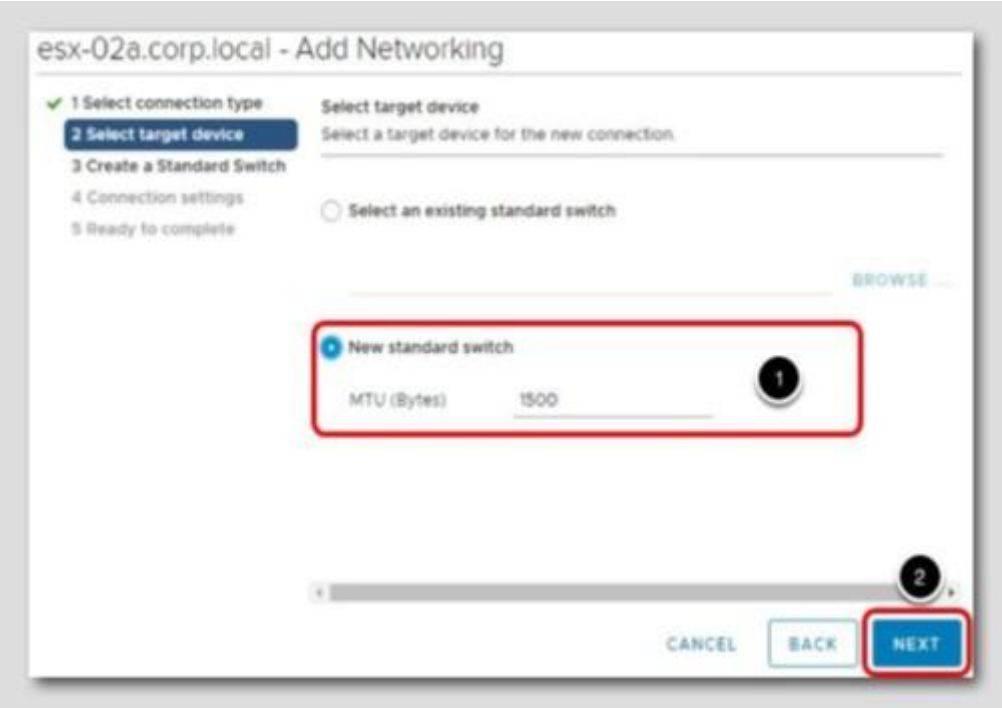
Connection Type



1. When asked to select connection type, choose Virtual Machine Port Group for a Standard Switch.
2. Click Next.

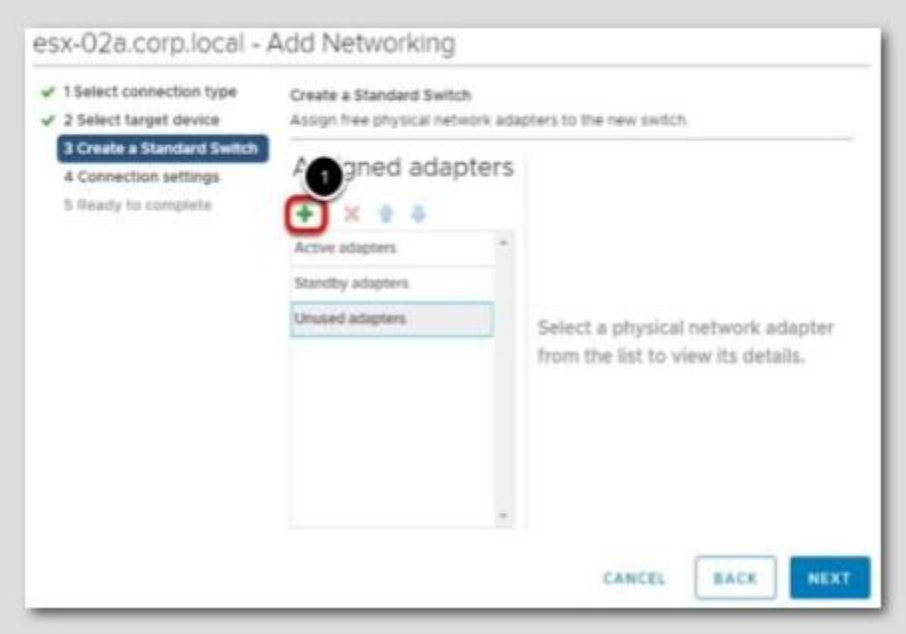
Target Device

1. When asked to select a target device, choose New Standard Switch. Note that a larger MTU size can be specified if needed.
2. Click Next.



Create a Standard Switch

1. Click the '+' button.

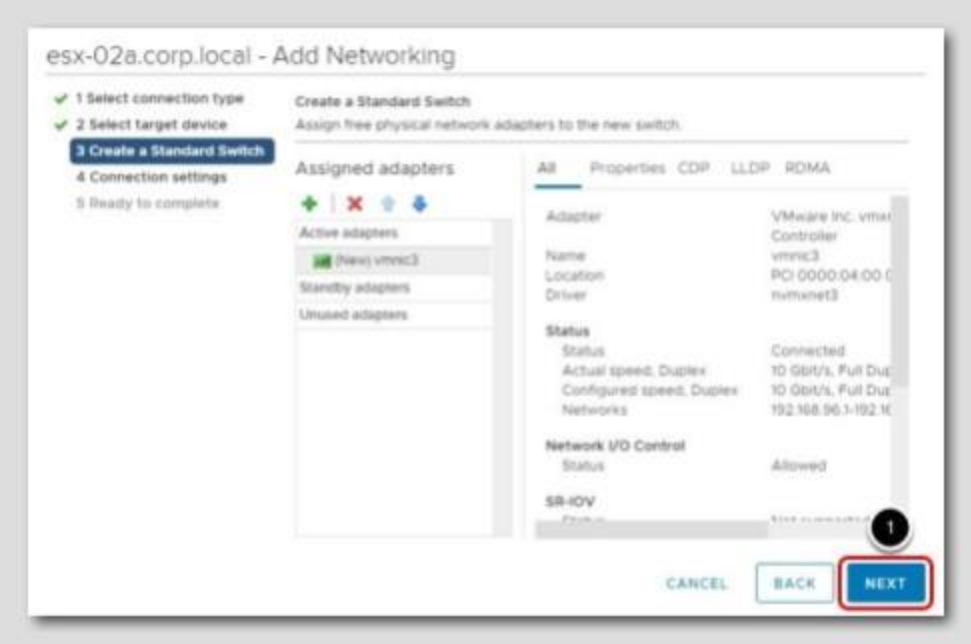


Add Physical Adapter

1. Select vmnic3 under Network Adapters
2. Click OK.

Add Physical Adapter

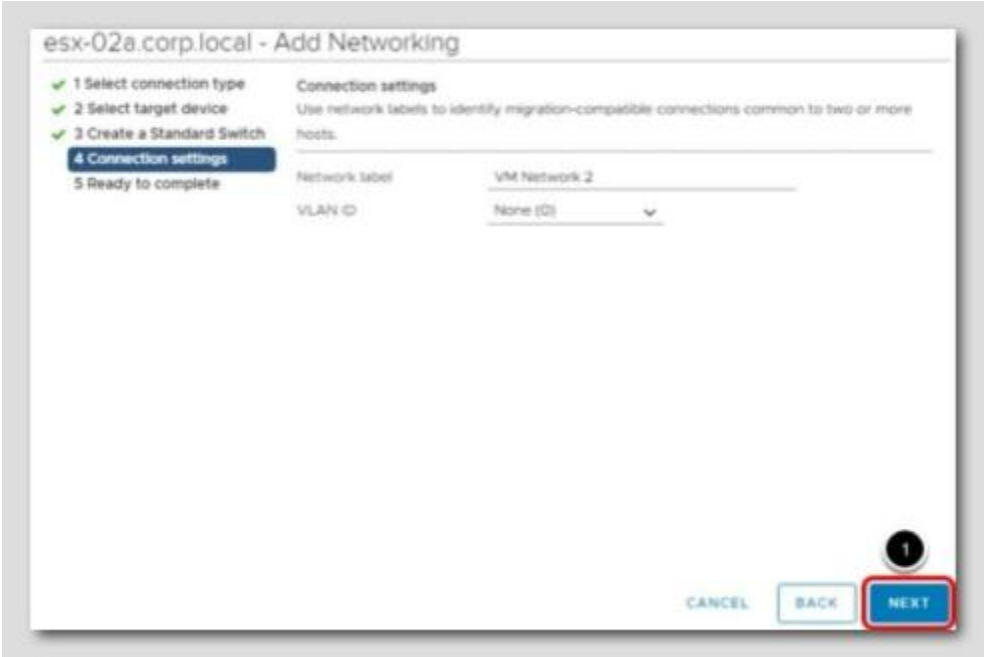
1. Click Next to continue.



Connection Settings

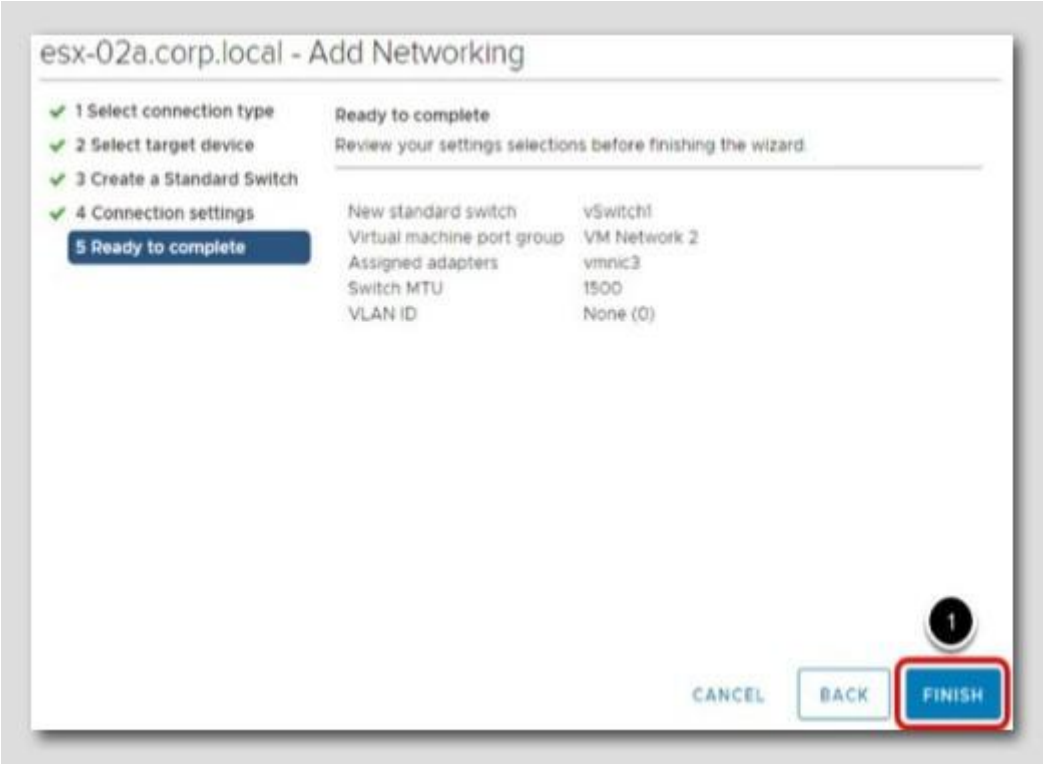
At the Connection settings step of the wizard, for Network label, leave the default name of VM Network 2. Do not change the VLAN ID; leave this set to None (0).

1. Click Next to continue.



Complete the Wizard

1. Review the port group settings in Ready to complete and click Finish.



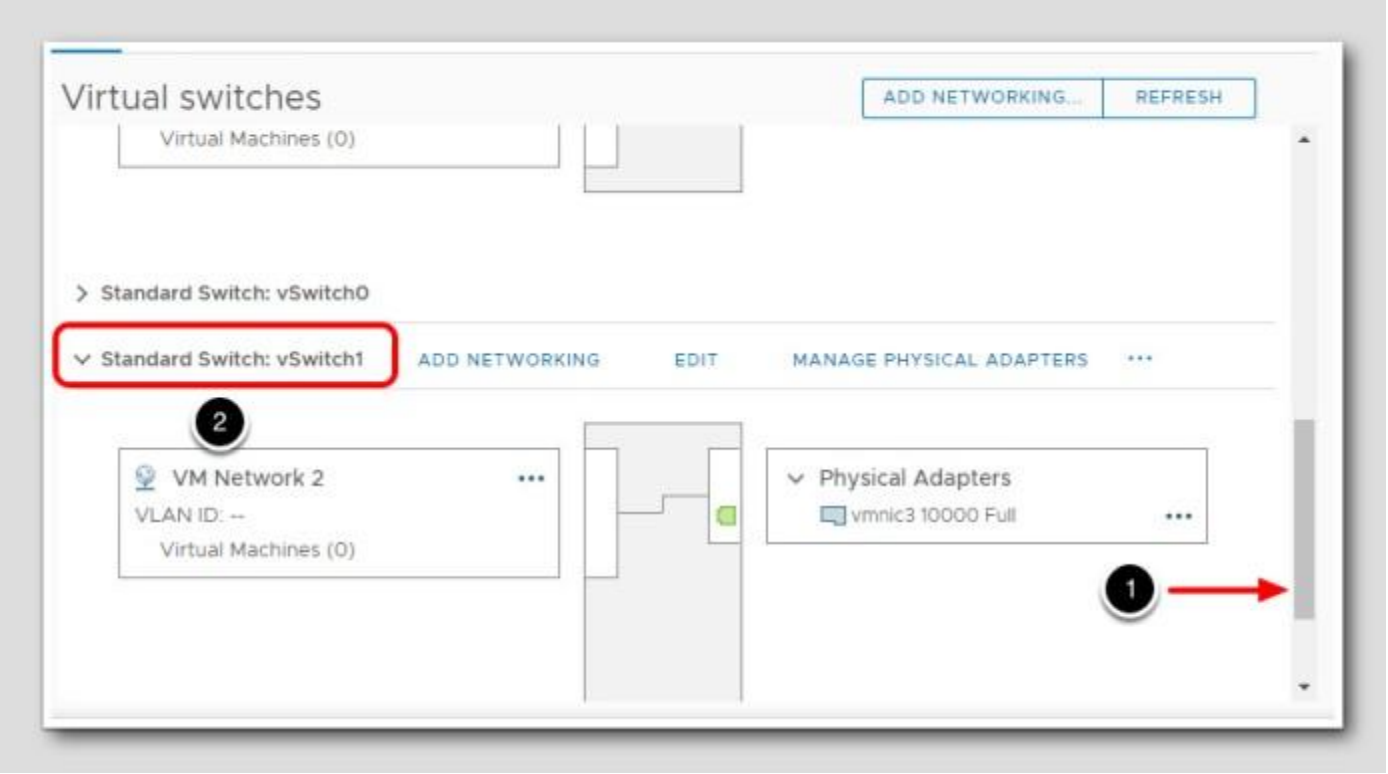
Virtual Switches

Next, we will verify the switch has been created.

1. Click Configure.
2. Click on Virtual Switches.

Standard Switch: vSwitch1

1. Scroll down until you see Standard Switch: vSwitch1.
2. If needed, expand the section.



Practical 3: Configure Access to an iSCSI datastore.

The vSphere Hypervisor, ESXi, provides host-level storage virtualization, which logically abstracts the physical storage layer from virtual machines.

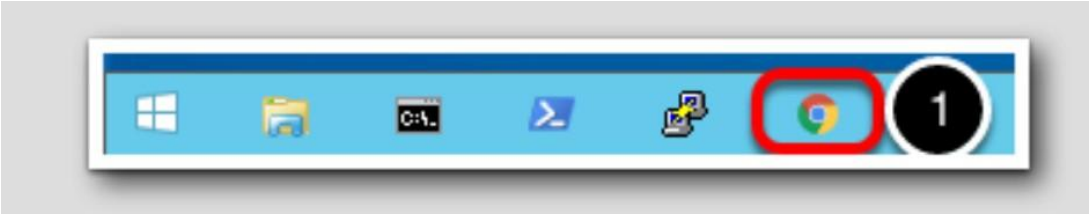
A vSphere virtual machine uses a virtual disk to store its operating system, program files, and other data associated with its activities. A virtual disk is a large physical file, or a set of files, that can be copied, moved, archived, and backed up as easily as any other file. You can configure virtual machines with multiple virtual disks.

To access virtual disks, a virtual machine uses virtual SCSI controllers. These virtual controllers include BusLogic Parallel, LSI Logic Parallel, LSI Logic SAS, and VMware Paravirtual. These controllers are the only types of SCSI controllers that a virtual machine can see and access.

Each virtual disk resides on a vSphere Virtual Machine File System (VMFS) datastore or an NFS-based datastore that are deployed on physical storage.

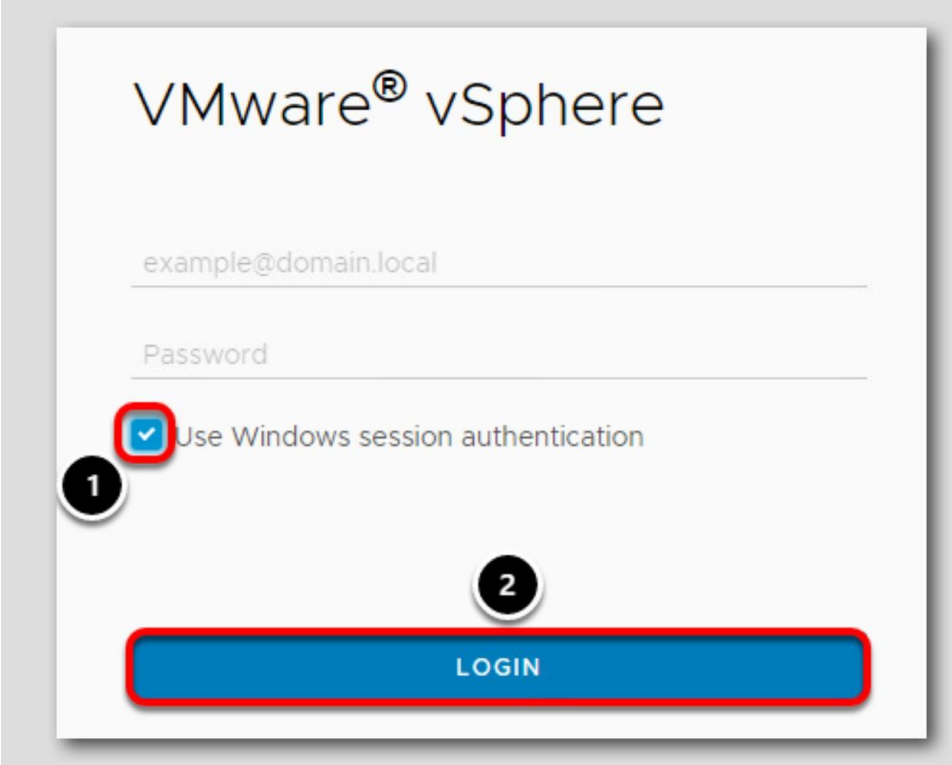
Launch Google Chrome web browser

1.Click on the Chrome Icon on the Windows Quick Launch Task Bar.



Enter credentials and log in

1. Select "Use Windows session authentication" check box.
2. Select Login.

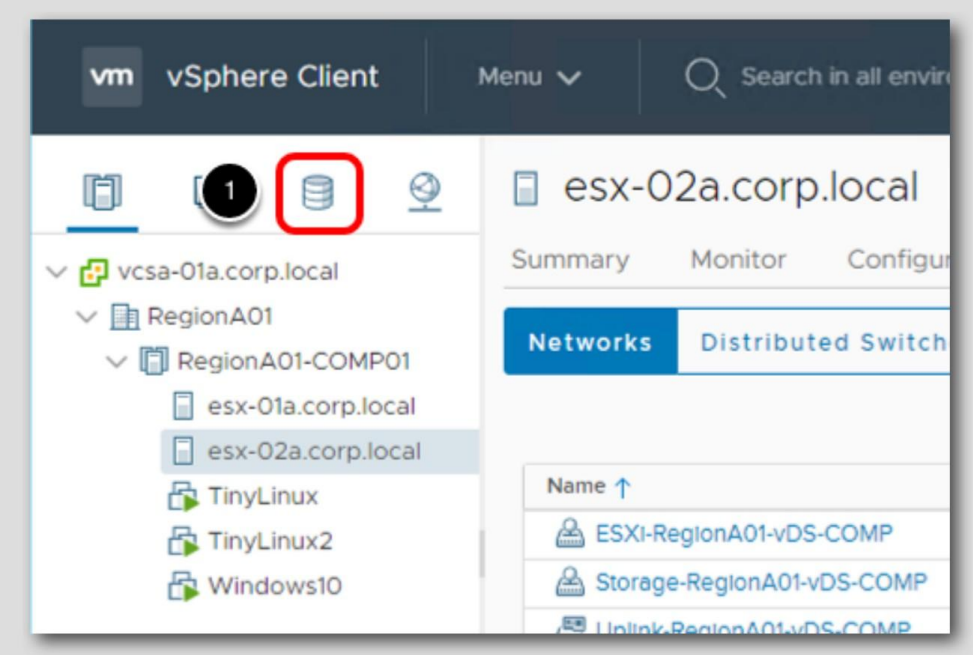


If credentials aren't saved, use the following:

- username: administrator@corp.local
- password: VMware1!

Navigate to Storage Management

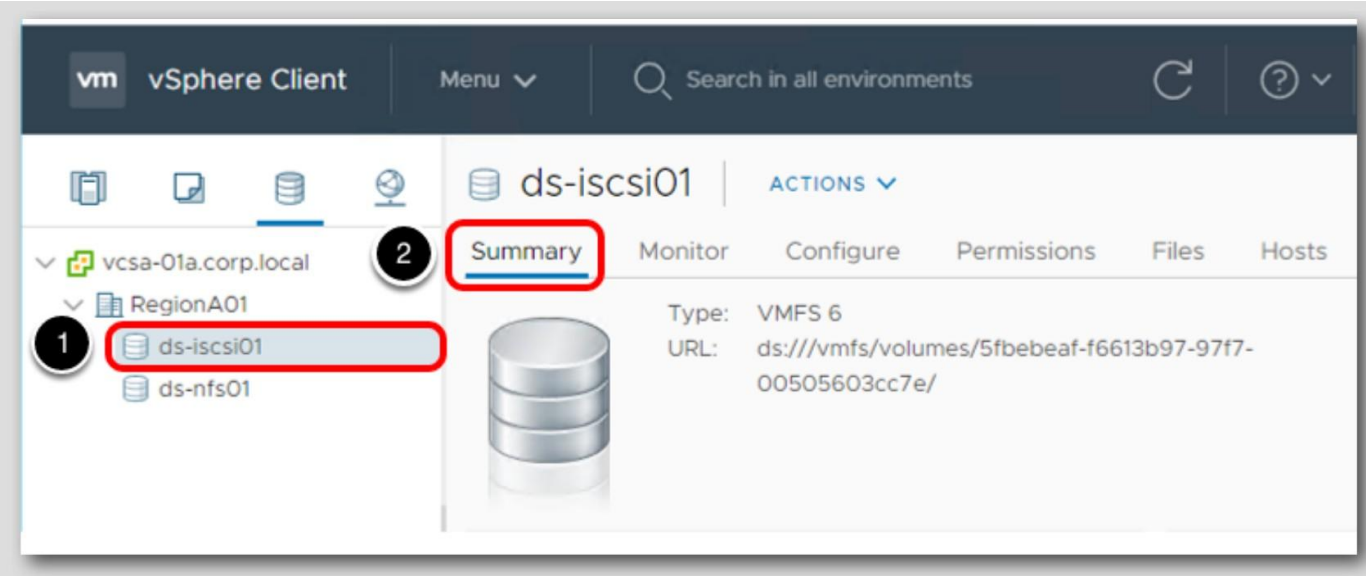
1.Select the Storage tab.



Expand RegionA01 Datacenter

There are 2 storage datastores configured, an ISCSI datastore and an NFS datastore.

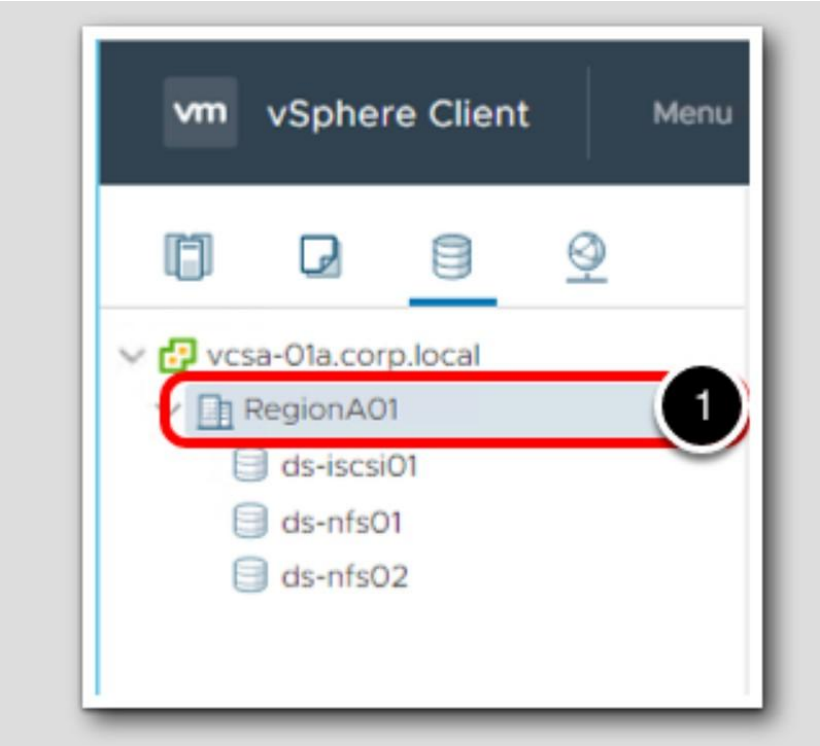
1. Select the ds-iscsi01 datastore.
2. Click on Summary for summary details of the datastore.



Practical 4: Create and manage VMFS datastore.

Create a VMFS Datastore

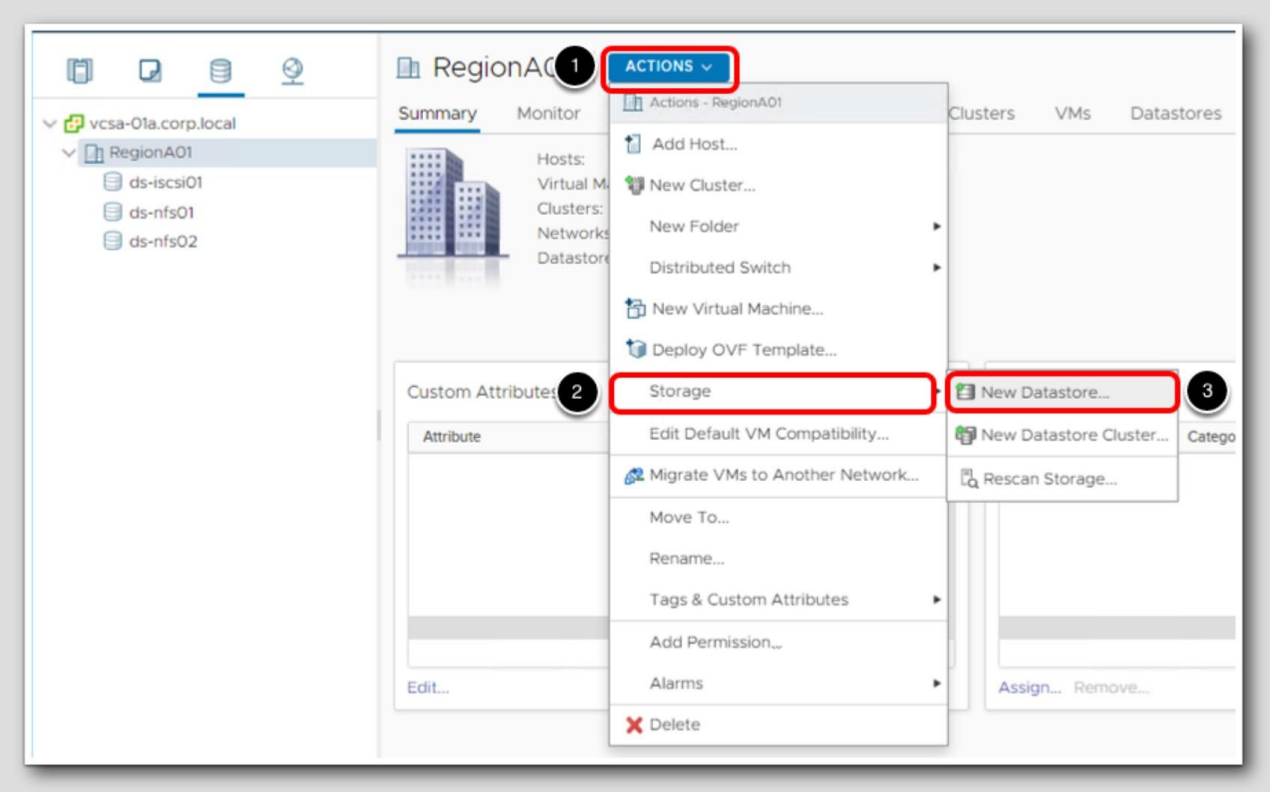
1.Select RegionA01 Datacenter.



New Datastore

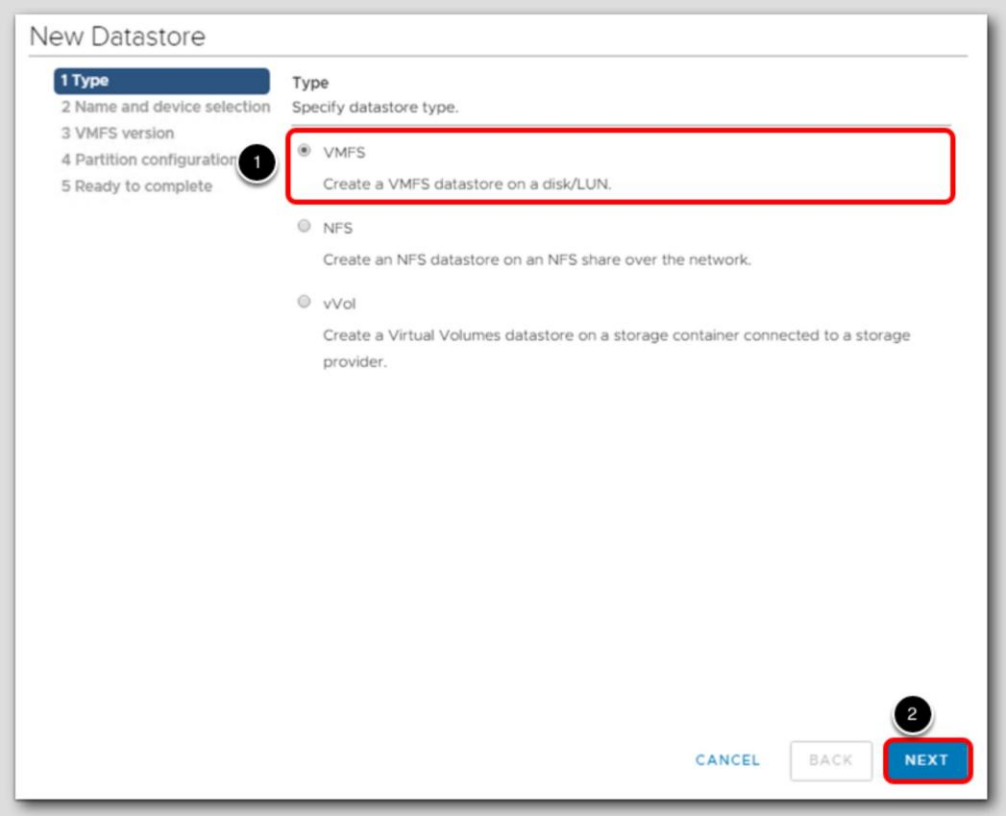
In this section, you will create a new vSphere iSCSI Datastore with a pre-provisioned iSCSI LUN.

1. Select Actions.
2. Select Storage.
3. Select New Datastore.



New Datastore - Type

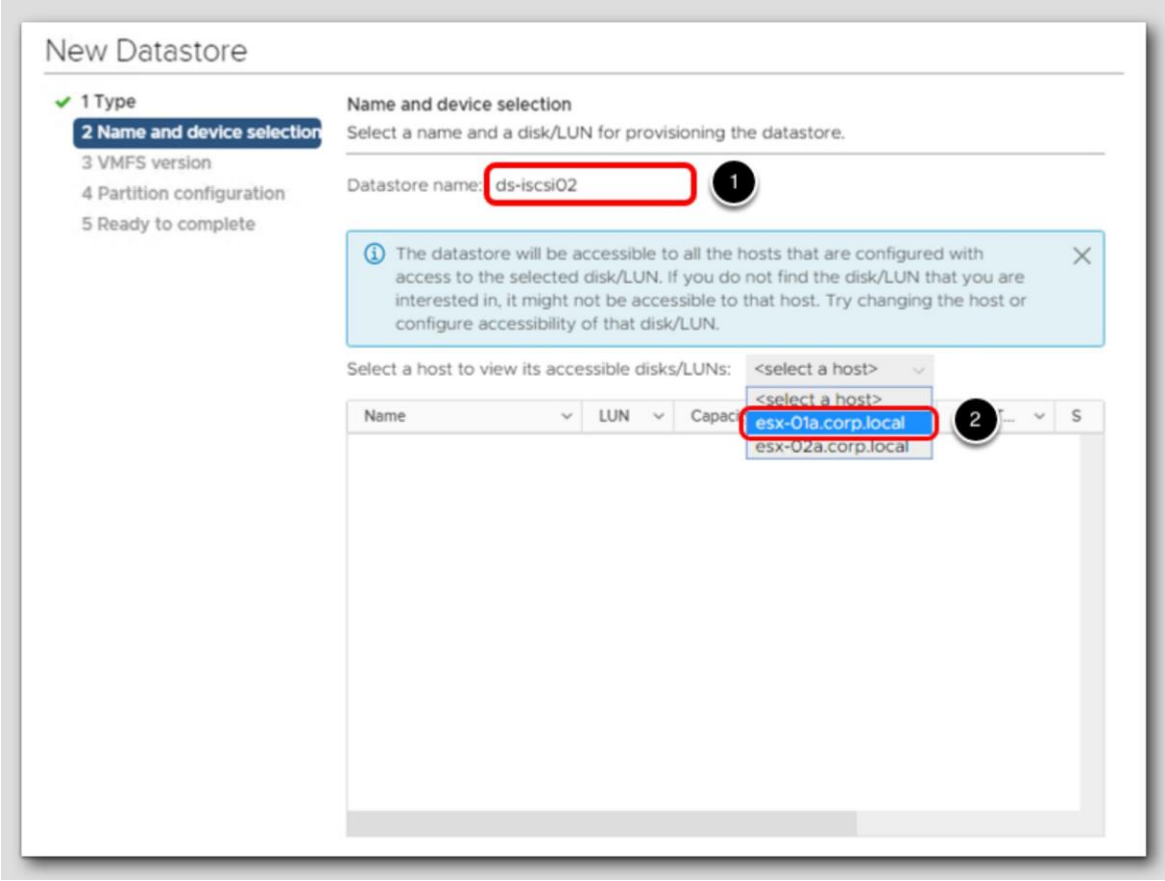
1. Verify VMFS is selected.
2. Click Next.



New Datastore - Name and Device configuration

1. Give the new Datastore the name ds-iscsi02.
2. Select a Host to view the accessible disks/LUNs and select esx-01a.corp.local in the drop-down box.

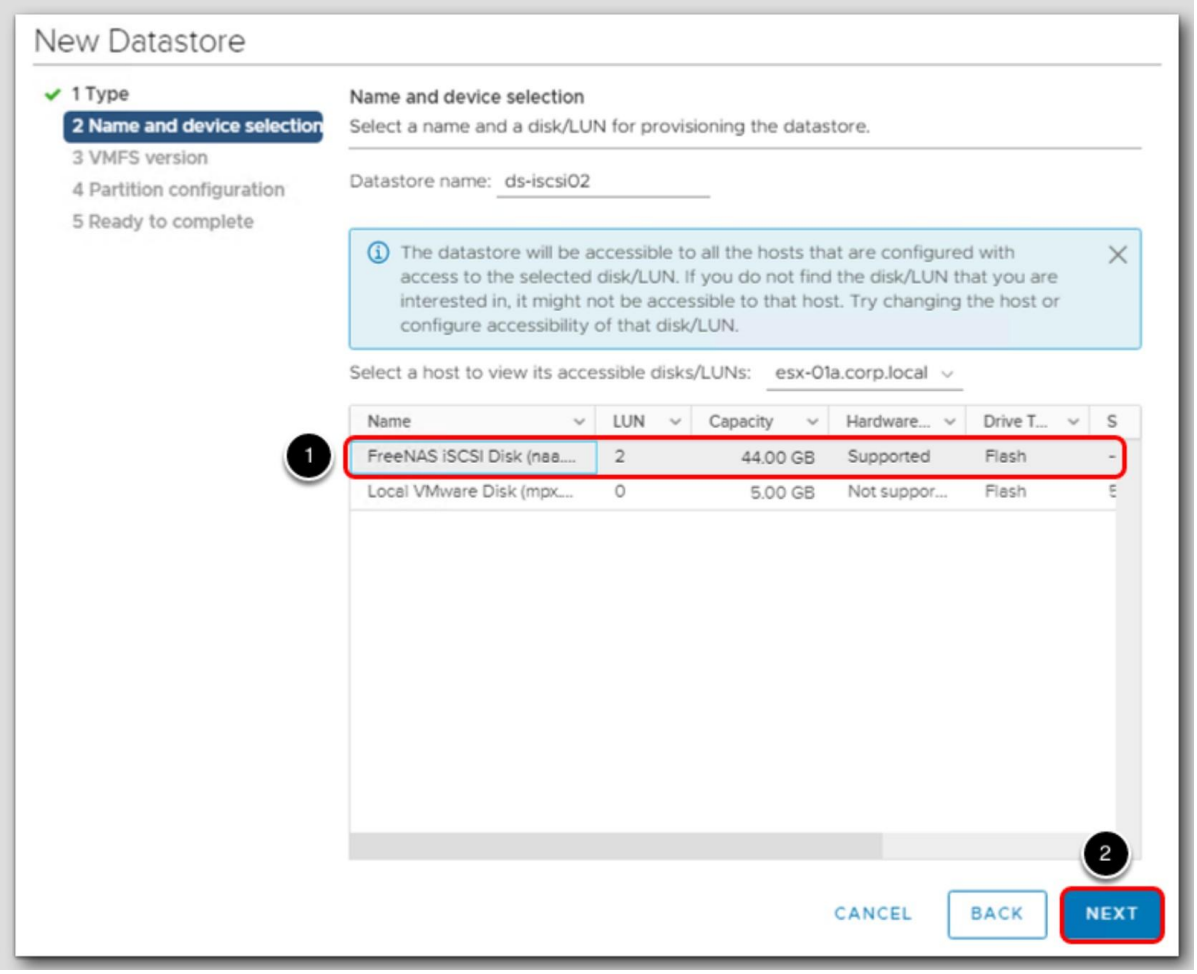
Note: Do not click Next just yet, proceed to the next step!



New Datastore - Name and device configuration (cont.)

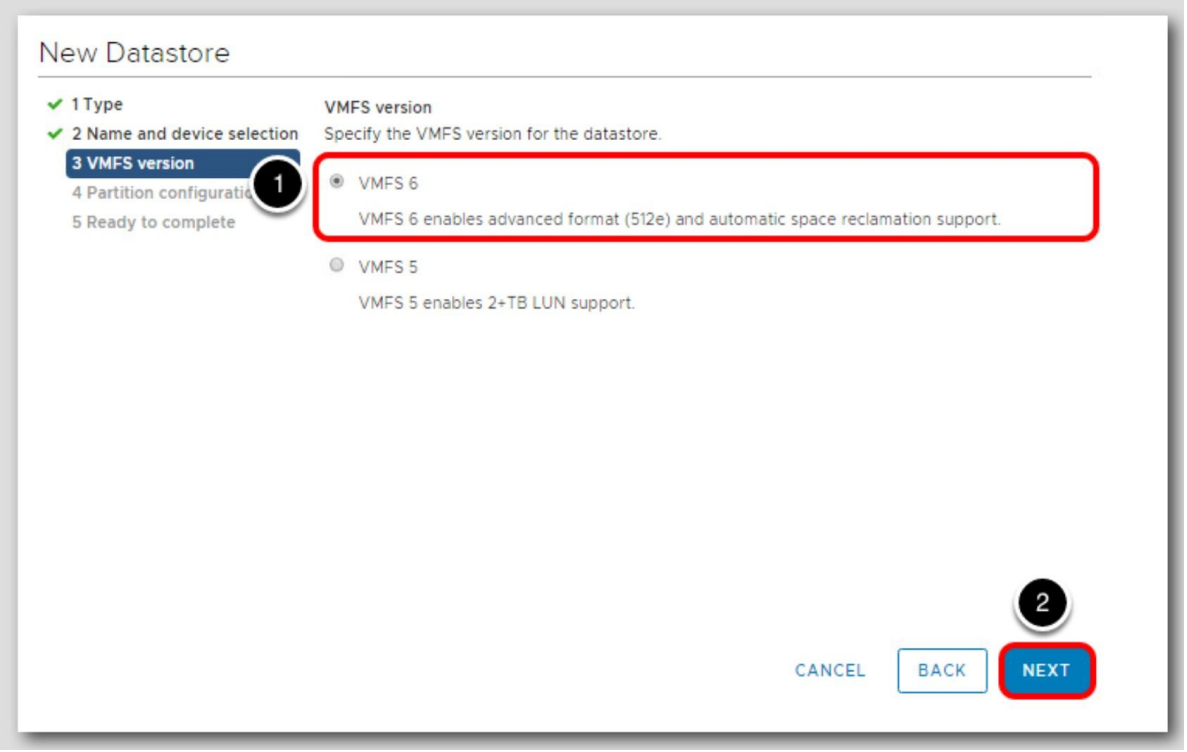
From this view, we can see that there are existing datastores that can be presented to our vSphere environment.

1. Select the device with LUN ID 2. In this case, it should be the only device visible with a FreeNAS prefix.
2. Click Next.



New Datastore - VMFS Version

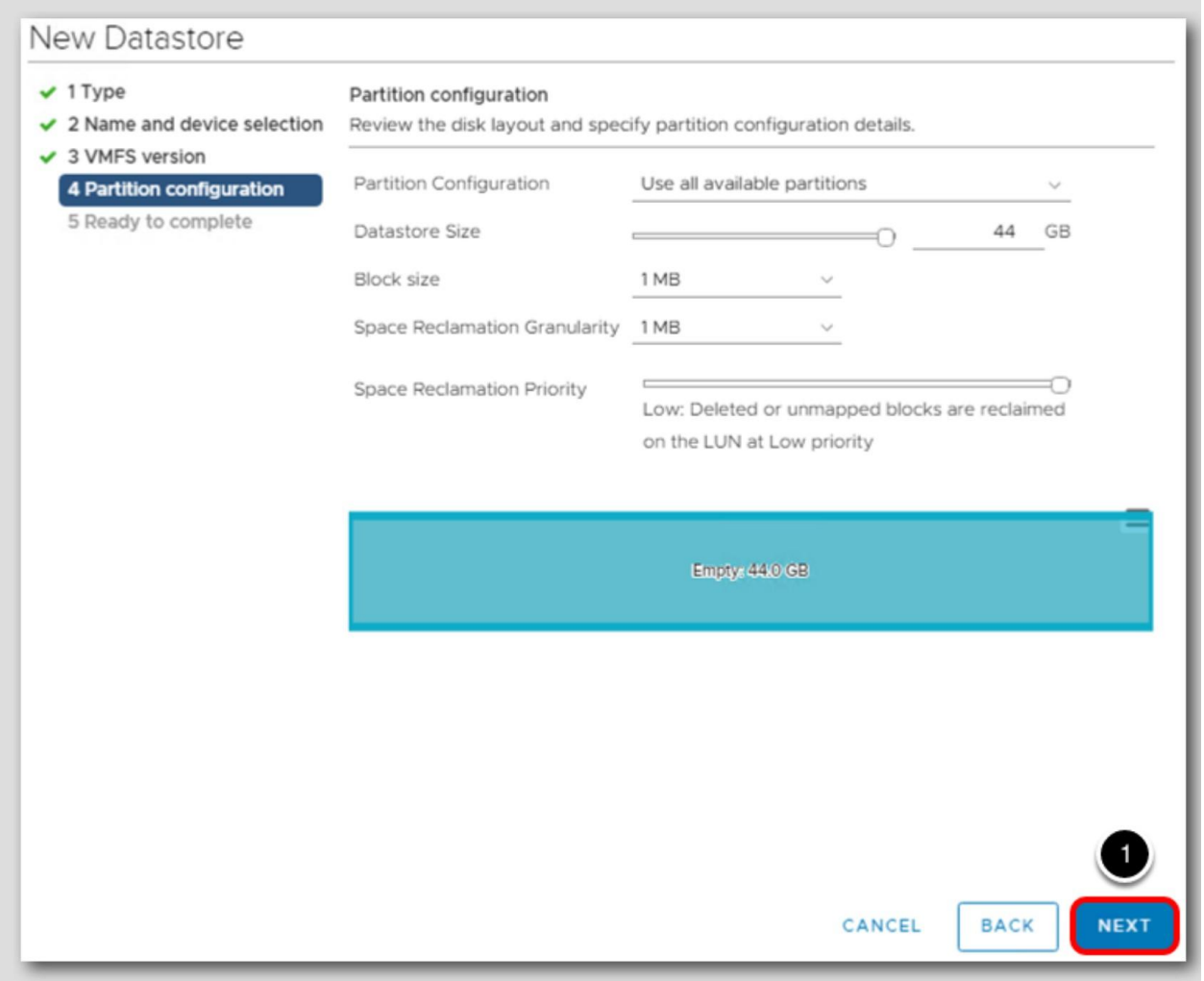
1. Leave the default of VMFS 6 selected.
2. Click Next.



New Datastore - Partition Configuration

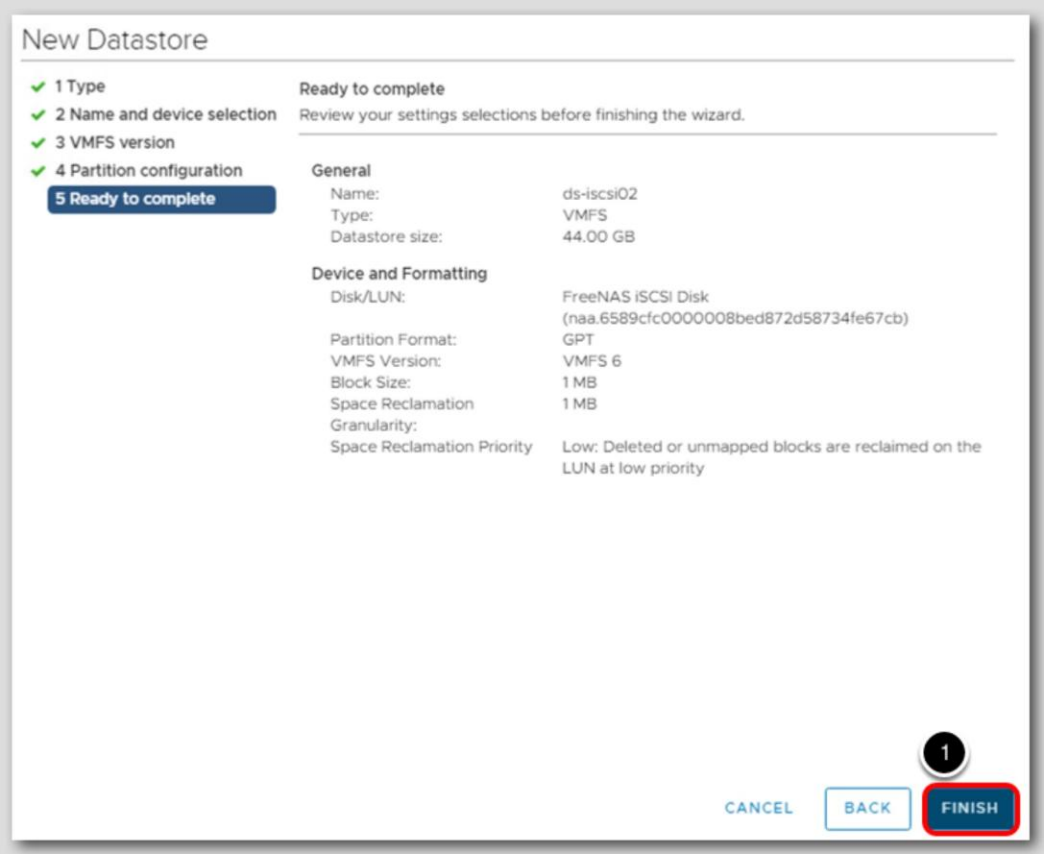
We can use all available capacity for this datastore or change the size if needed. The defaults are fine for this step.

1. Select Next.



New Datastore - Ready to complete

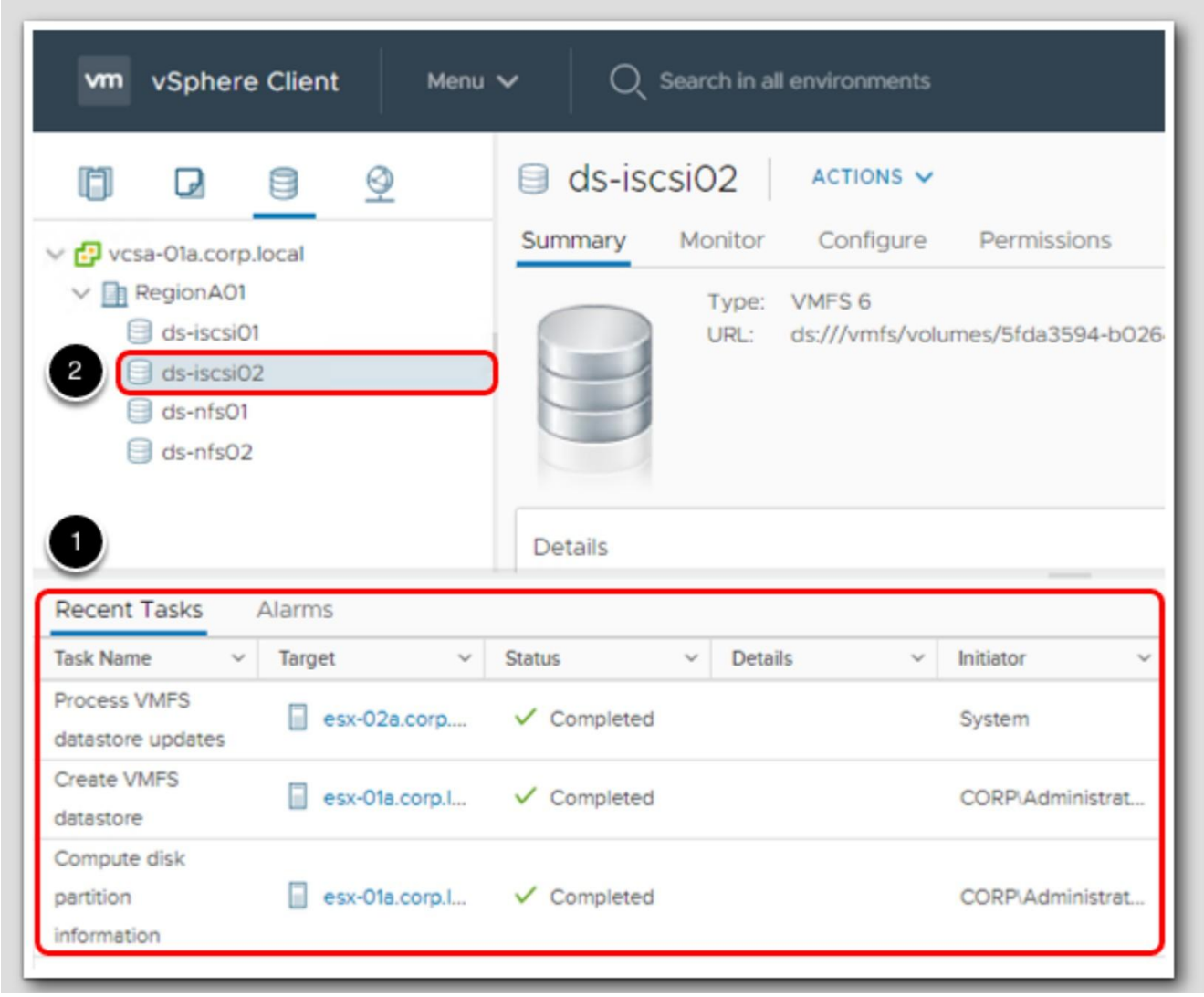
1. Review New Datastore configuration and click Finish.



New Datastore - Monitor task progress

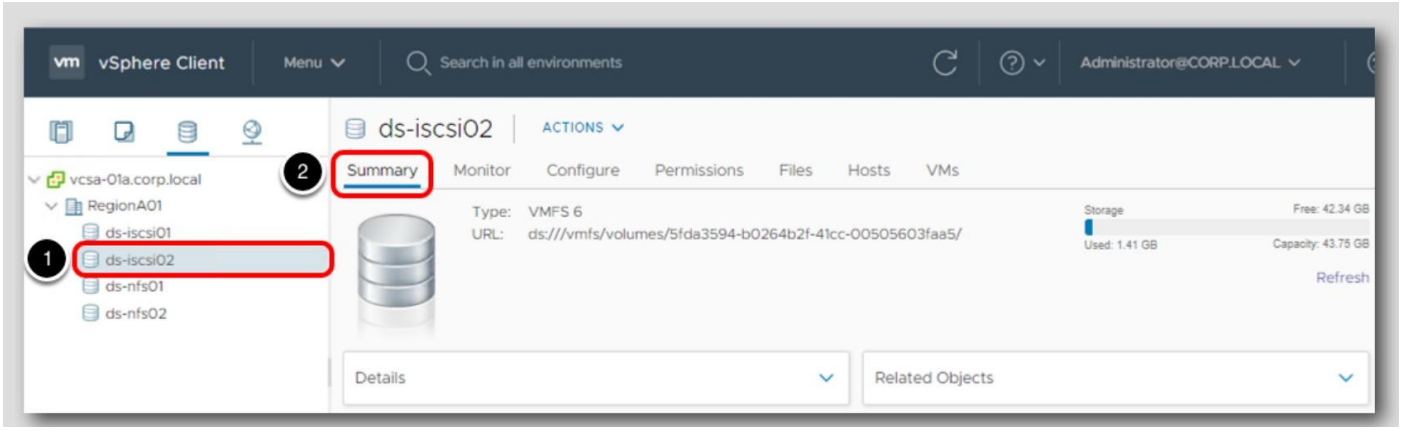
1. Note the progress in the Recent Tasks pane.

2.When complete, you should see the ds-iscsi02 Datastore available for use.



New Datastore - Review Settings

1. Select the datastore ds-iscsi02 from the inventory list
2. Select Summary to review capacity and configuration details



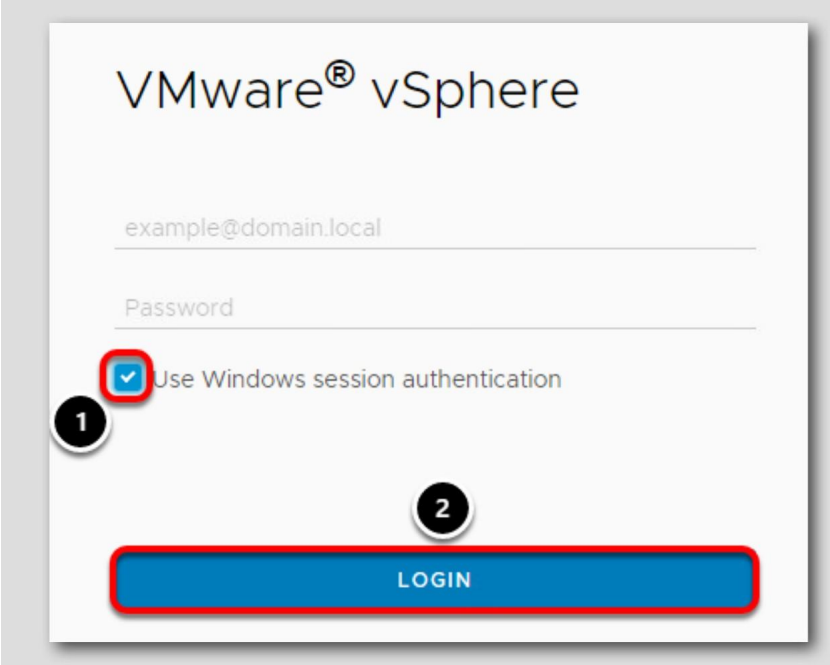
Practical 5: Configure Access to an NFS datastore.

Launch Google Chrome web browser

1.Click on the Chrome Icon on the Windows Quick Launch Task Bar.

Enter credentials and log in

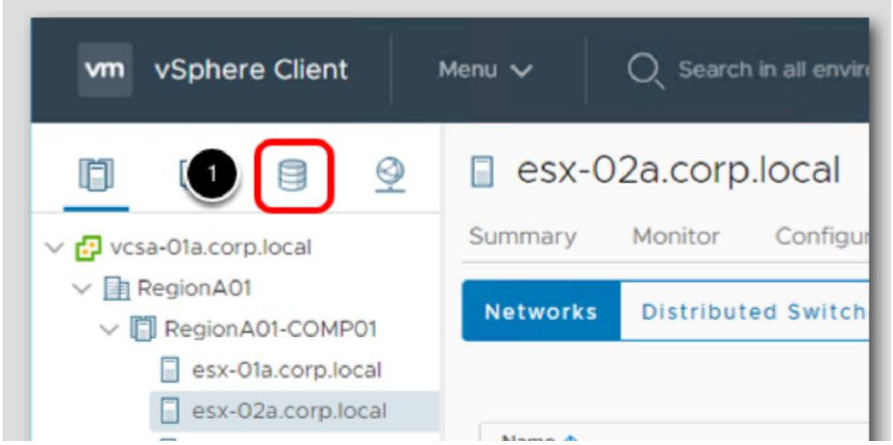
1. Select "Use Windows session authentication" check box.
2. Select Login.



If credentials aren't saved, use the following: ● username: administrator@corp.local ● password: VMware1!

Navigate to Storage Management

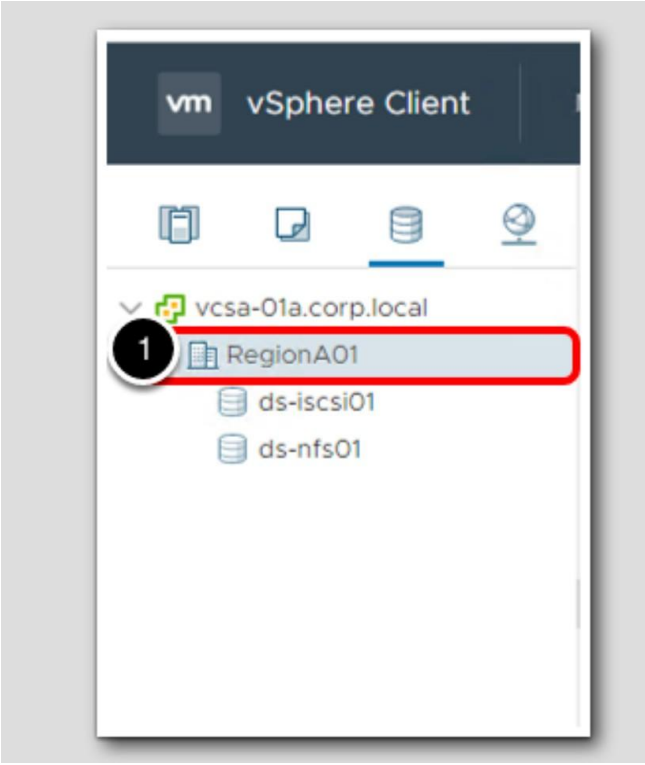
1.Select the Storage tab.



Create a vSphere NFS Datastore

In this section, you will create a new vSphere NFS Datastore using a pre-provisioned NFS mount.

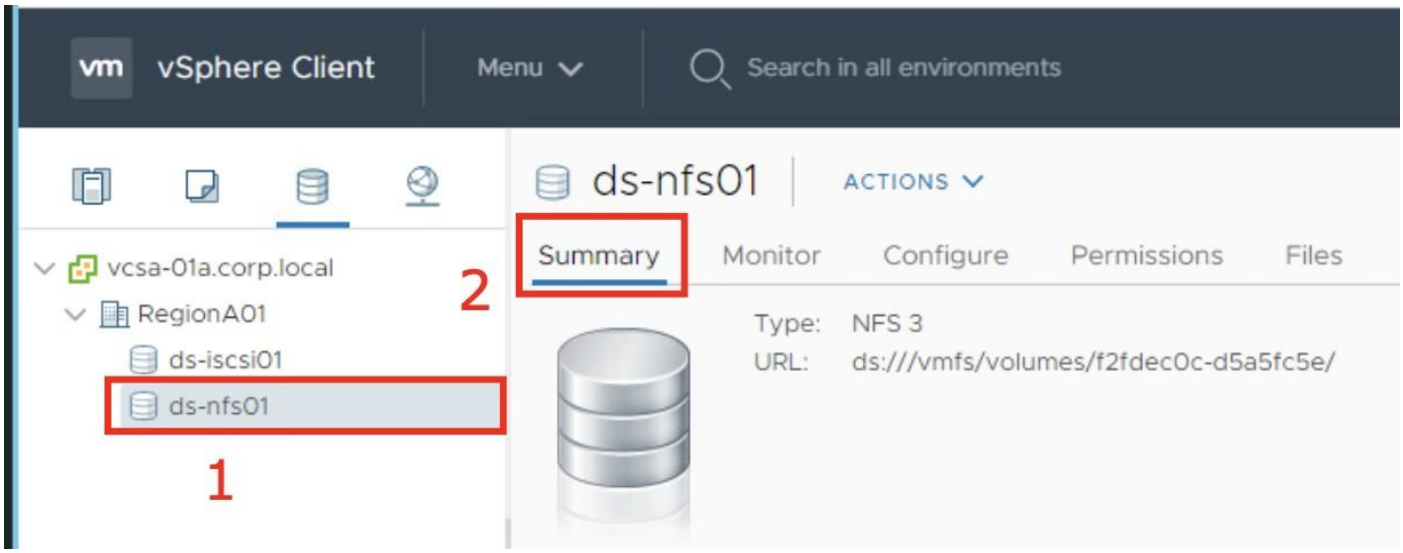
1. Select RegionA01 Datacenter



Expand RegionA01 Datacenter

There are 2 storage datastores configured, an ISCSI datastore and an NFS datastore.

1. Select the ds-nfs01 datastore.
2. Click on Summary for summary details of the datastore.


vSphere Client
Menu
Search in all environments
ds-nfs01
ACTIONS
Summary
Monitor
Configure
Permissions
Files
vcsa-01a.corp.local
RegionA01
ds-iscsi01
ds-nfs01
1
2
Type: NFS 3
URL: ds:///vmfs/volumes/f2fdec0c-d5a5fc5e/

Practical 6: Deploy a new virtual machine from a template and clone a virtual machine.

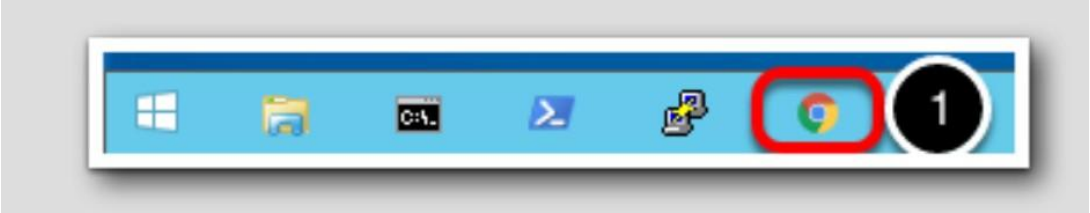
VMware provides several ways to provision vSphere virtual machines.

Cloning a virtual machine can save time if you are deploying many similar virtual machines. You can create, configure, and install software on a single virtual machine. You can clone it multiple times, rather than creating and configuring each virtual machine individually.

Another provisioning method is to clone a virtual machine to a template. A template is a master copy of a virtual machine that you can use to create and provision virtual machines. Creating a template can be useful when you need to deploy multiple virtual machines from a single baseline but want to customize each system independently of the next. A common value point for using templates is to save time. If you have a virtual machine that you will clone frequently, make that virtual machine a template, and deploy your virtual machines from that template.

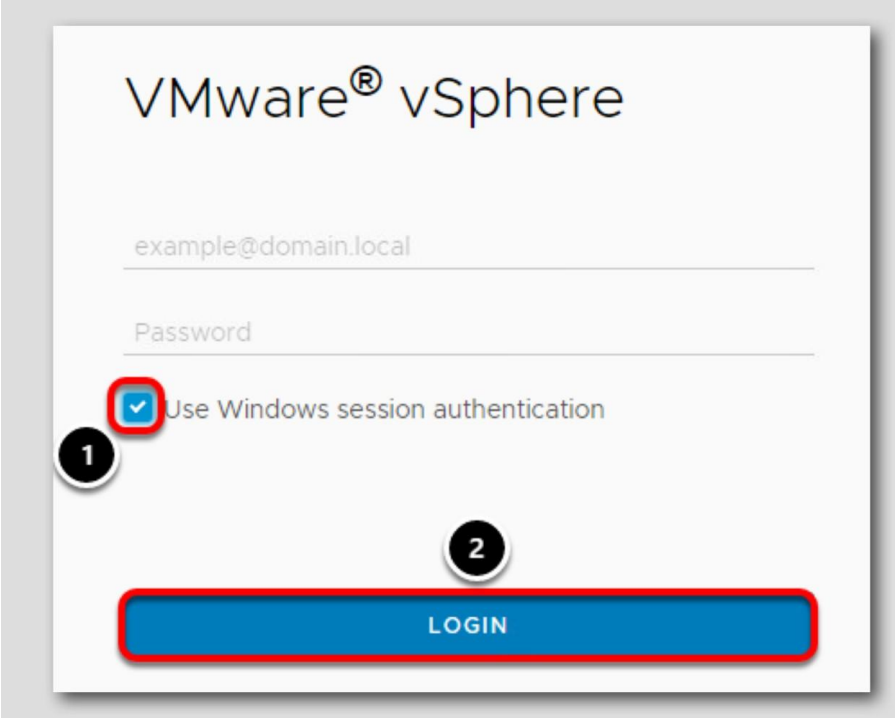
Launch Google Chrome web browser

1.Click on the Chrome Icon on the Windows Quick Launch Task Bar.



Enter credentials and log in

1. Select "Use Windows session authentication" check box.
2. Select Login.



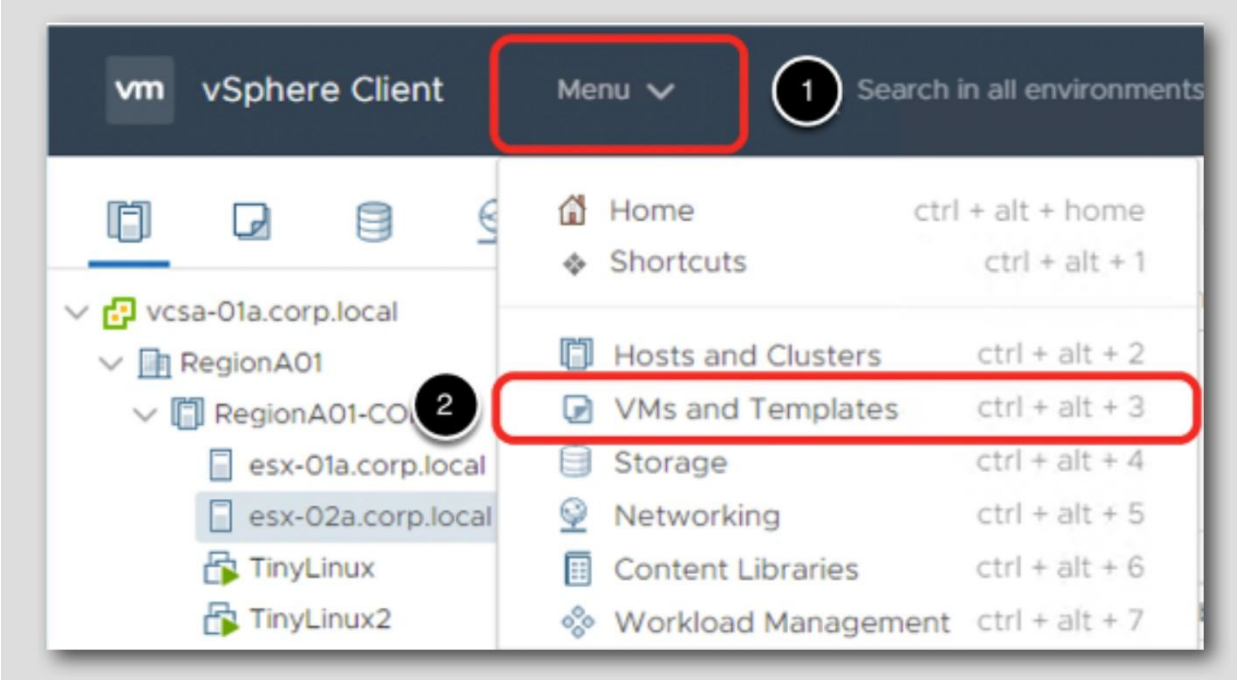
If credentials aren't saved, use the following:

- username: administrator@corp.local
- password: VMware1!

Navigate to the VMs and Templates management pane

1. Click on Menu.

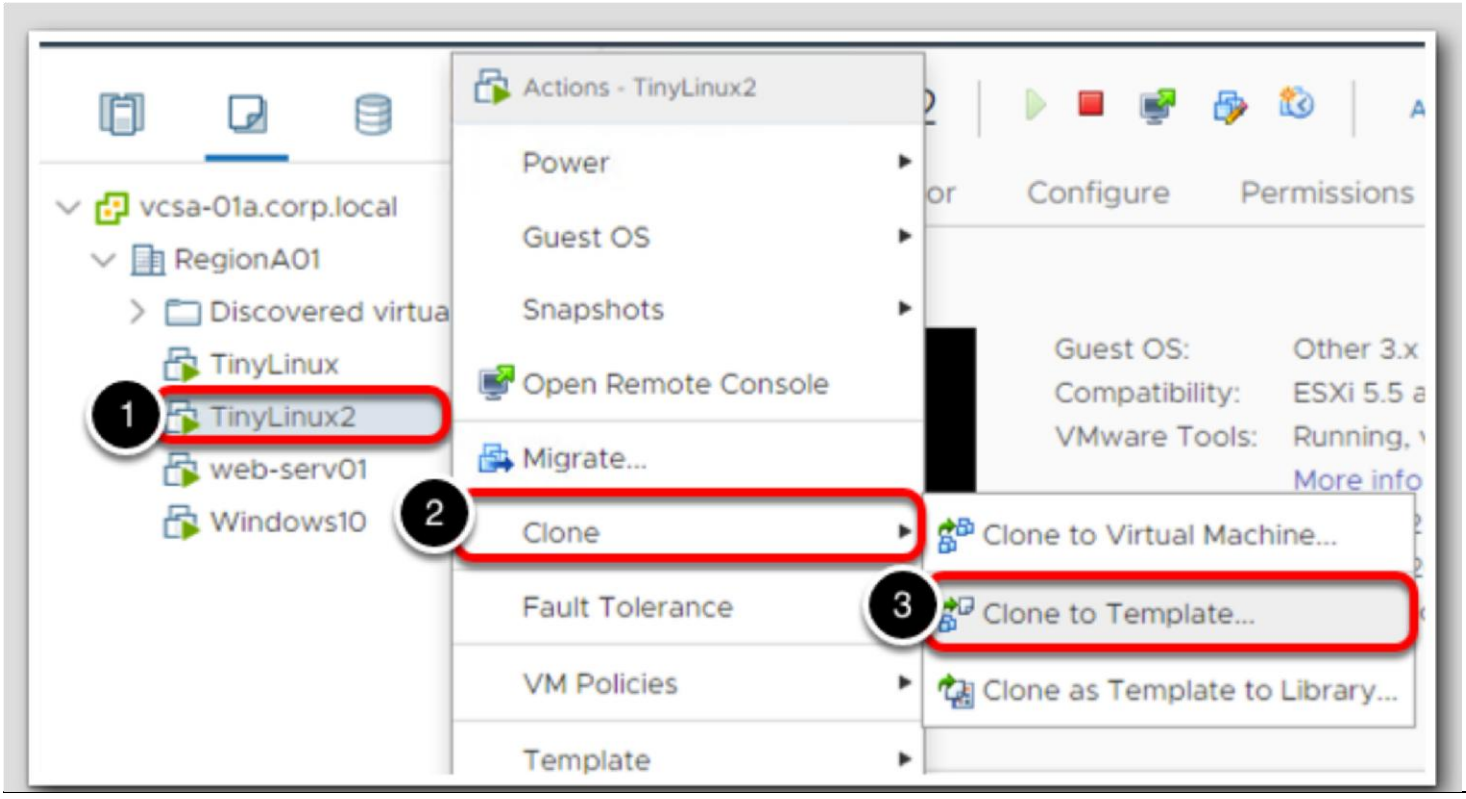
2. Select VMs and Templates.



Launch the Clone Virtual Machine to Template wizard 1. Right-click the Virtual Machine TinyLinux2.

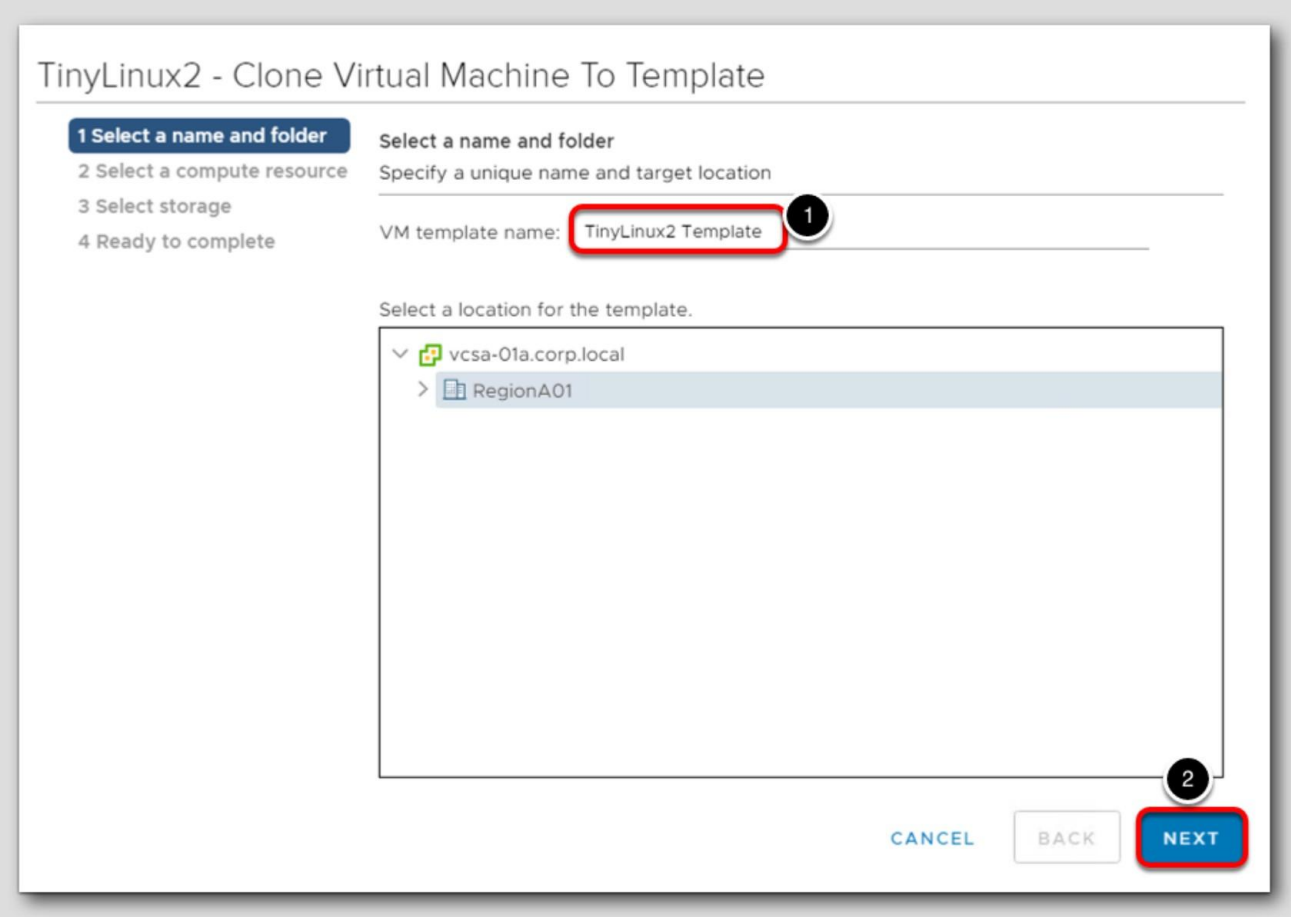
2. Select Clone.
3. Select Clone to Template…

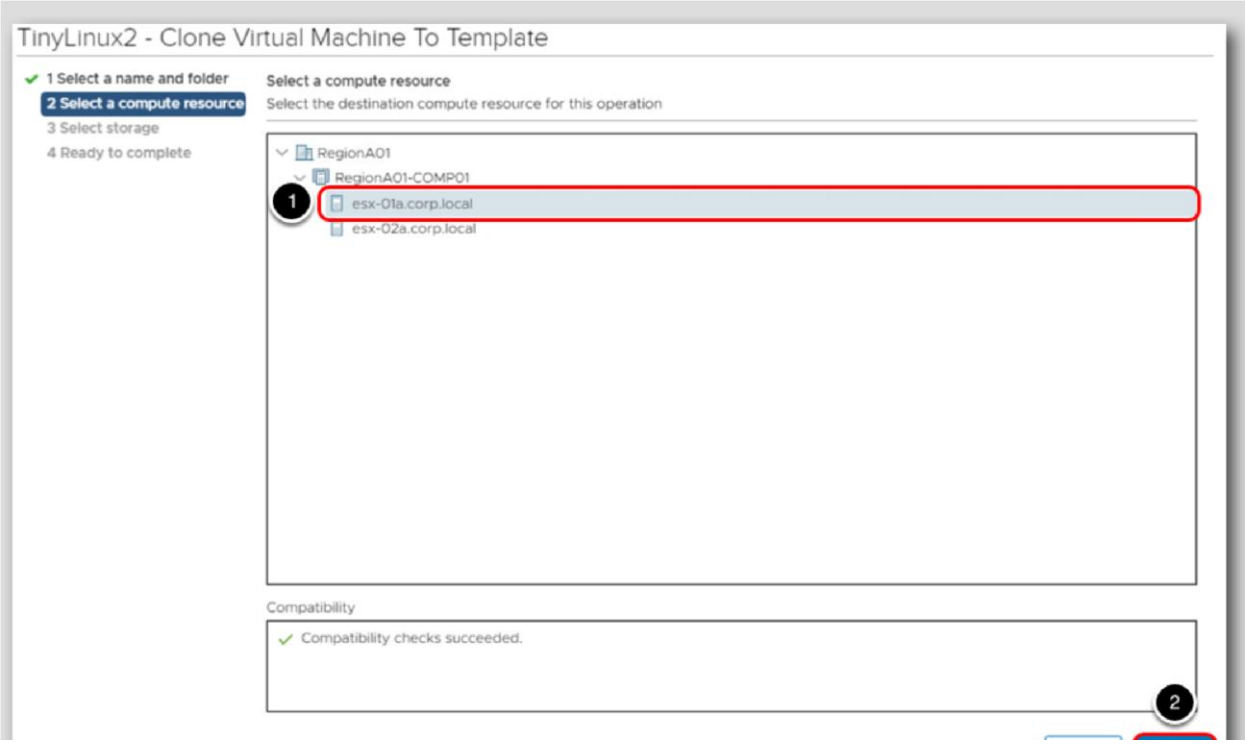
Select a name and folder



In the Clone Virtual Machine to Template wizard, provide a name for the Template - TinyLinux2 Template.

1. Please leave the location as RegionA01 for this lab.
2. Click Next



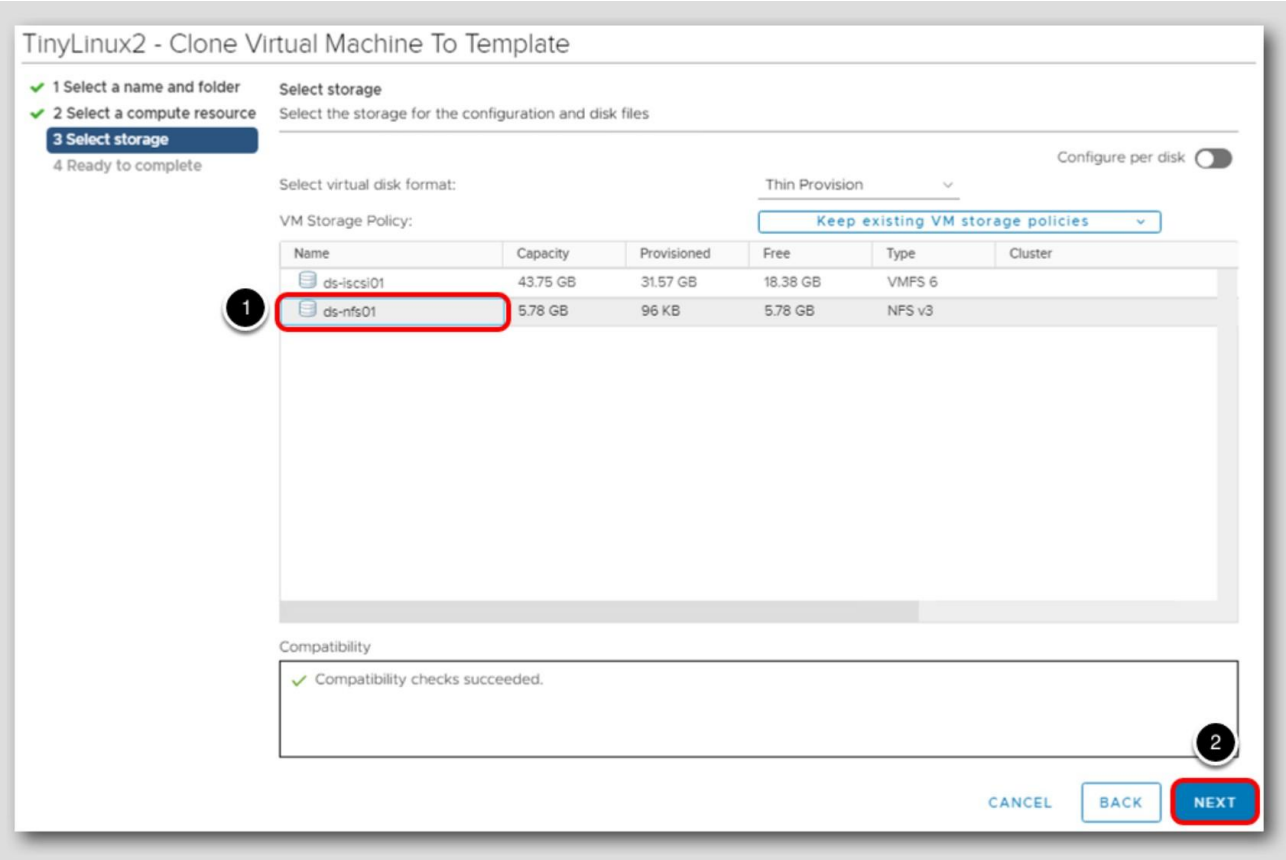


Select Compute Resource

1. Choose esx-01a.corp.local.
2. Click Next.

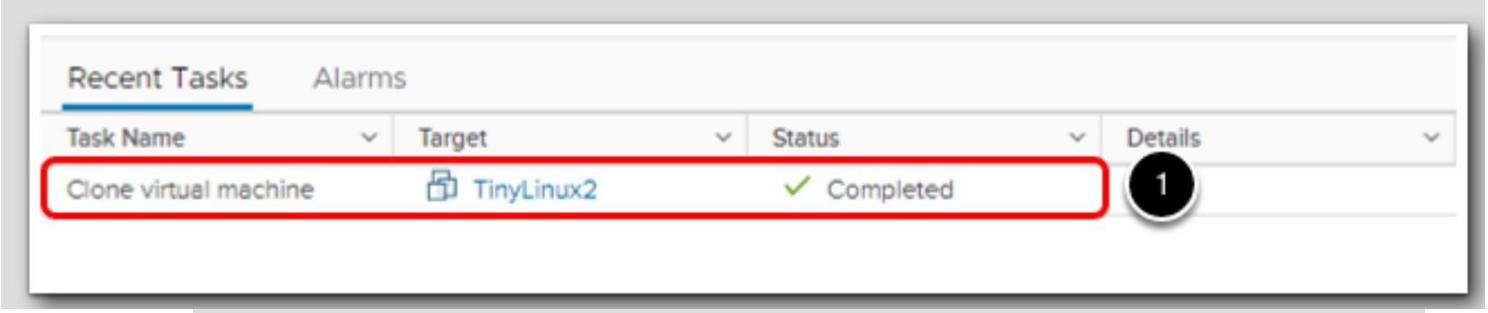
Select Storage

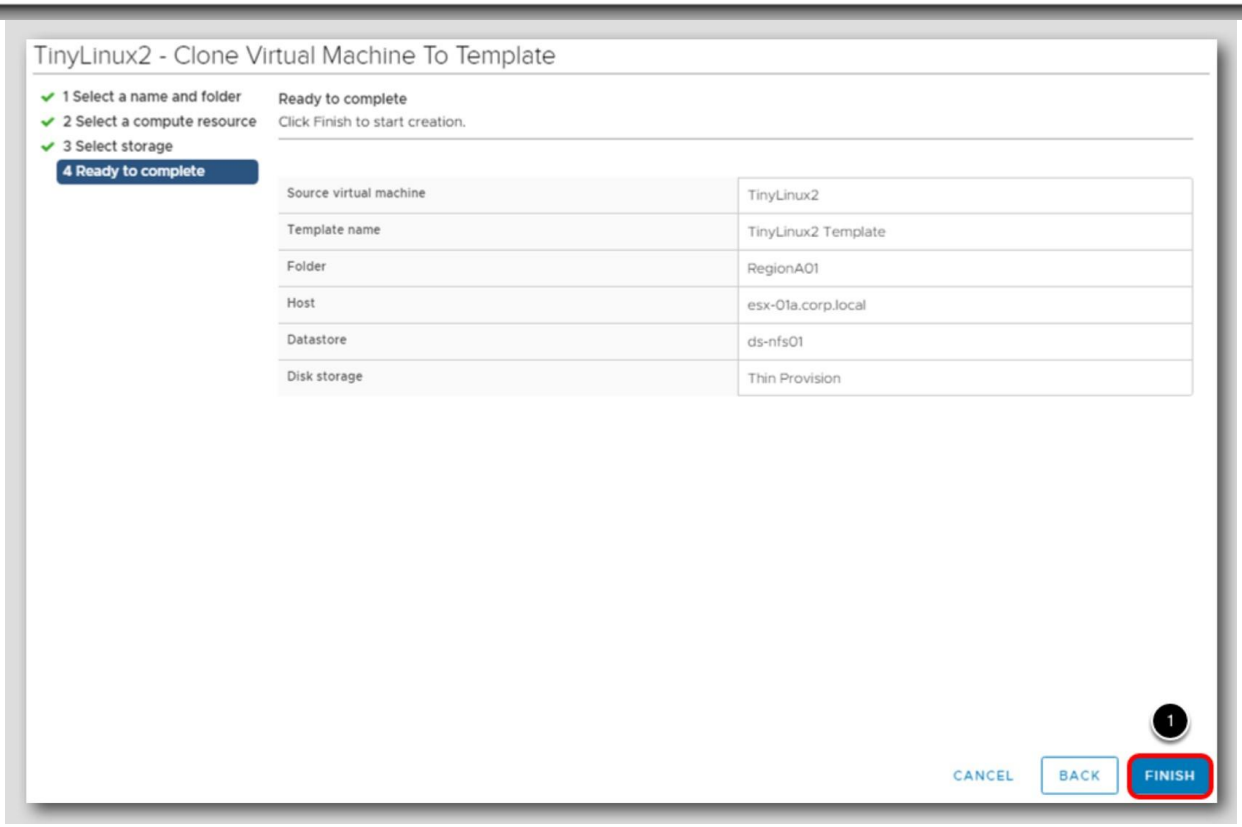
1. Select ds-nfs01 as the datastore.
2. Press the Next button



Review the VM Template Settings

1.Review the VM Template settings and press the Finish button.

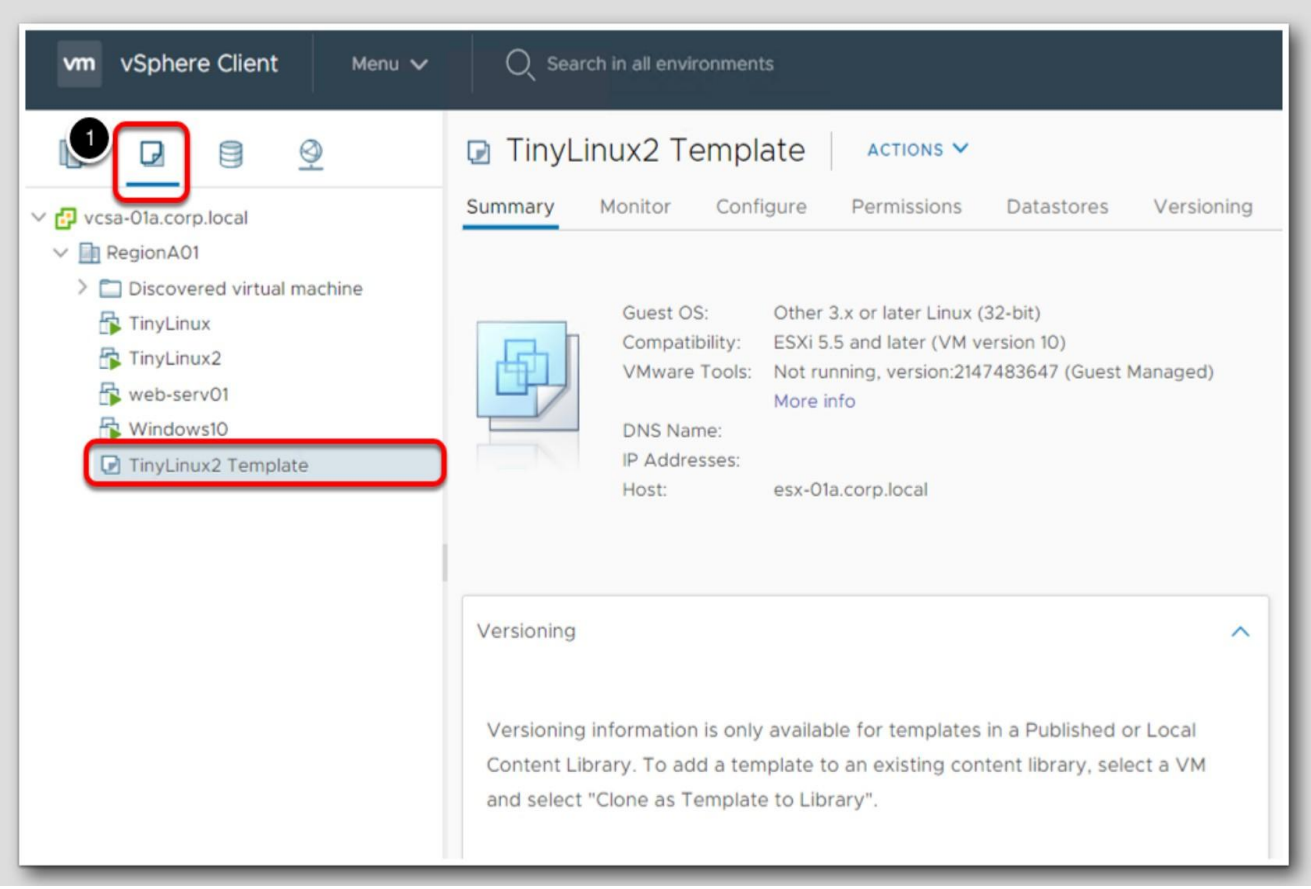




Monitor task progress

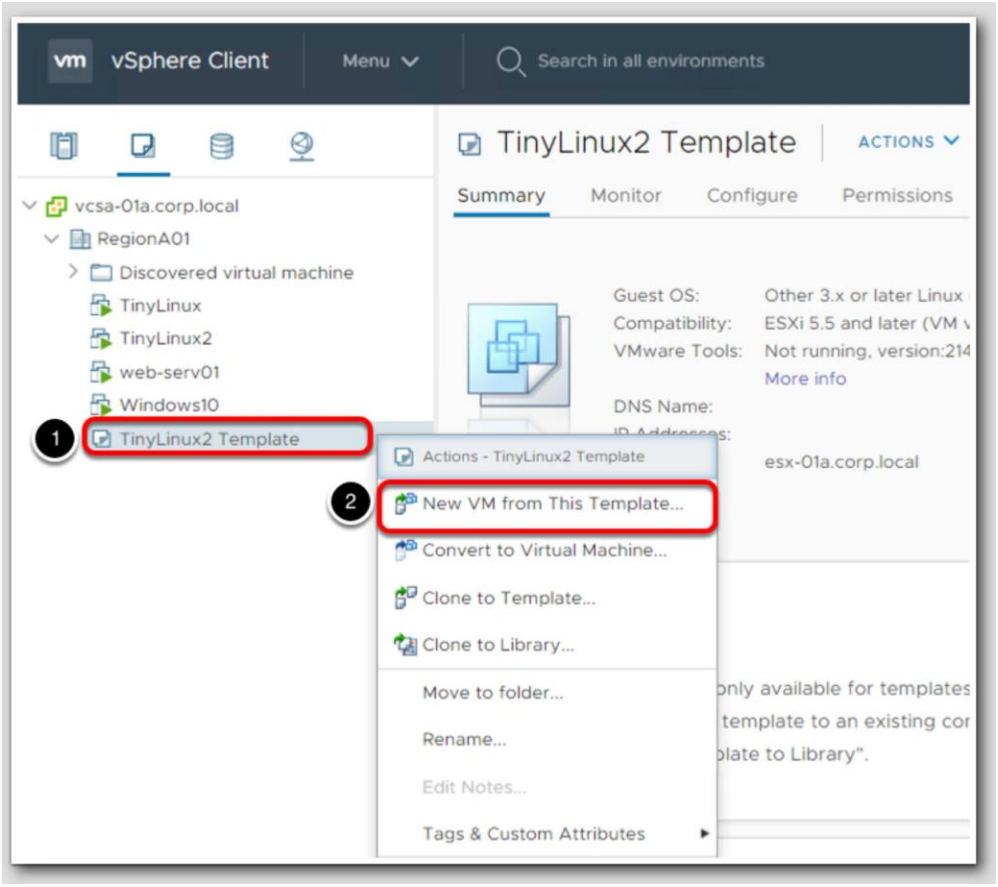
1. You can monitor the progress in the recent task window.

1. Once the task has been completed, click on the VM and TemplatesVM and Templates icon.
   TinyLinux 2 Template object should be on the inventory pane.



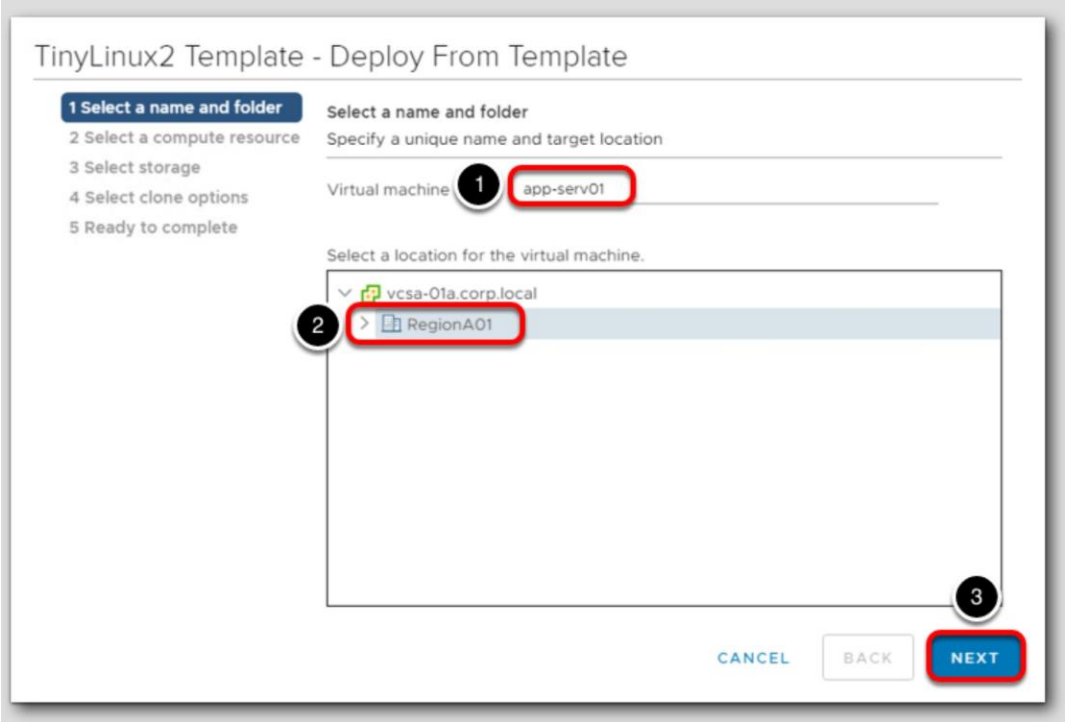
Launch the Deploy From Template wizard

1. Select the Template, TinyLinux2 Template
2. Right click on TinyLinux2 Template and select New VM from This Template.



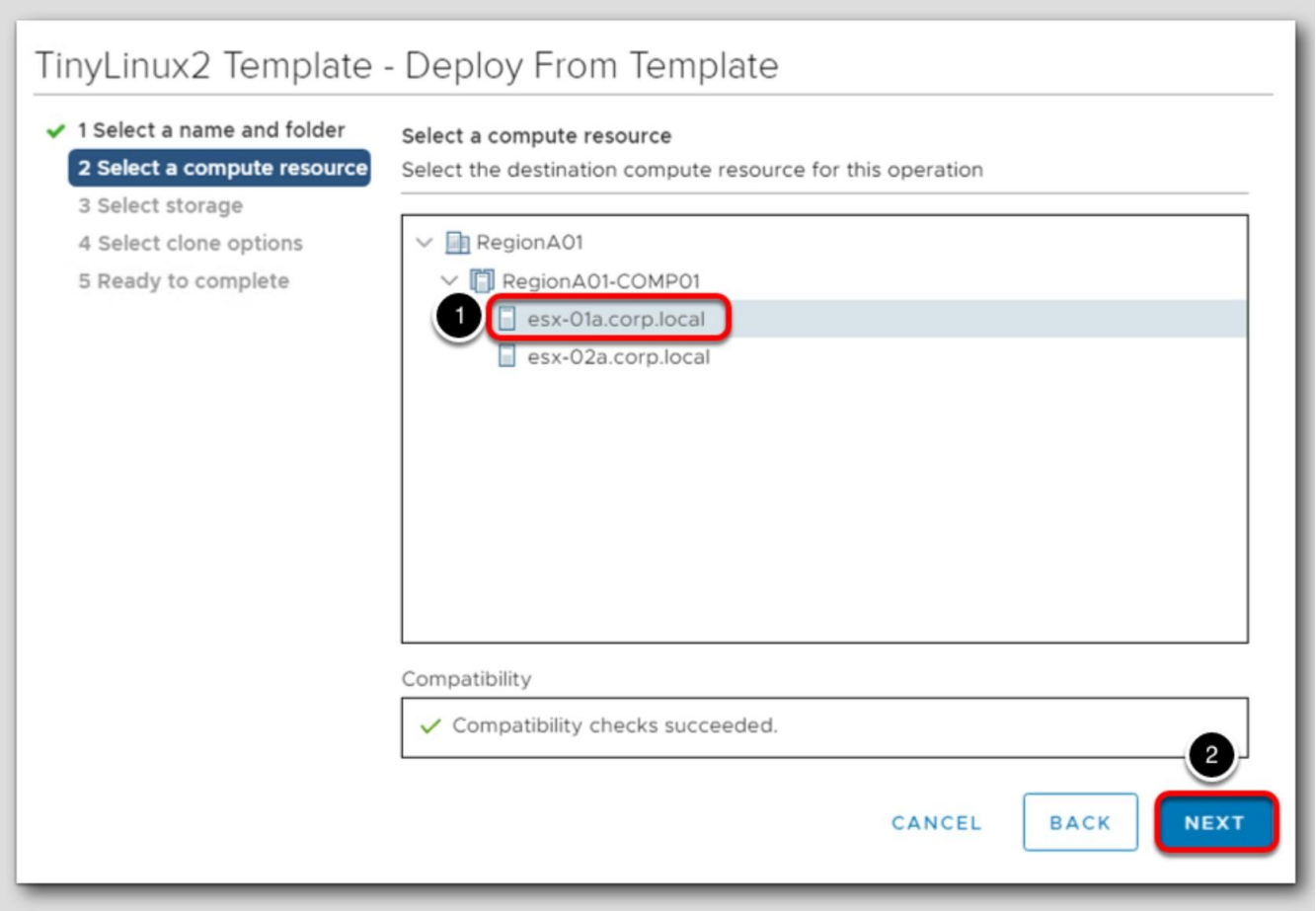
Select a name and folder

1. Enter app-serv01 for the name of the new virtual machine.
2. Leave the default location of RegionA01 Datacenter.
3. Click the Next button.



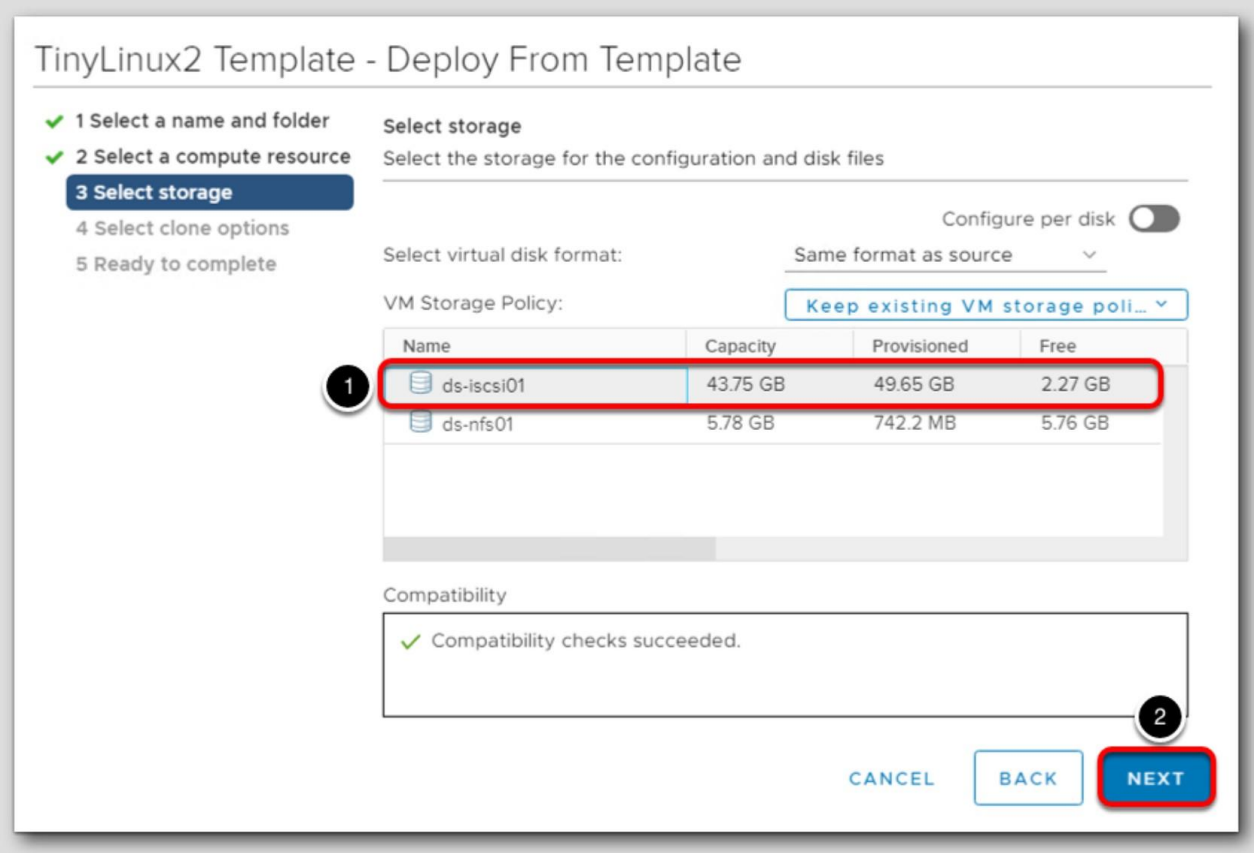
Select compute resource

1. Select esx-01a.corp.local.
2. Click Next.



Select storage

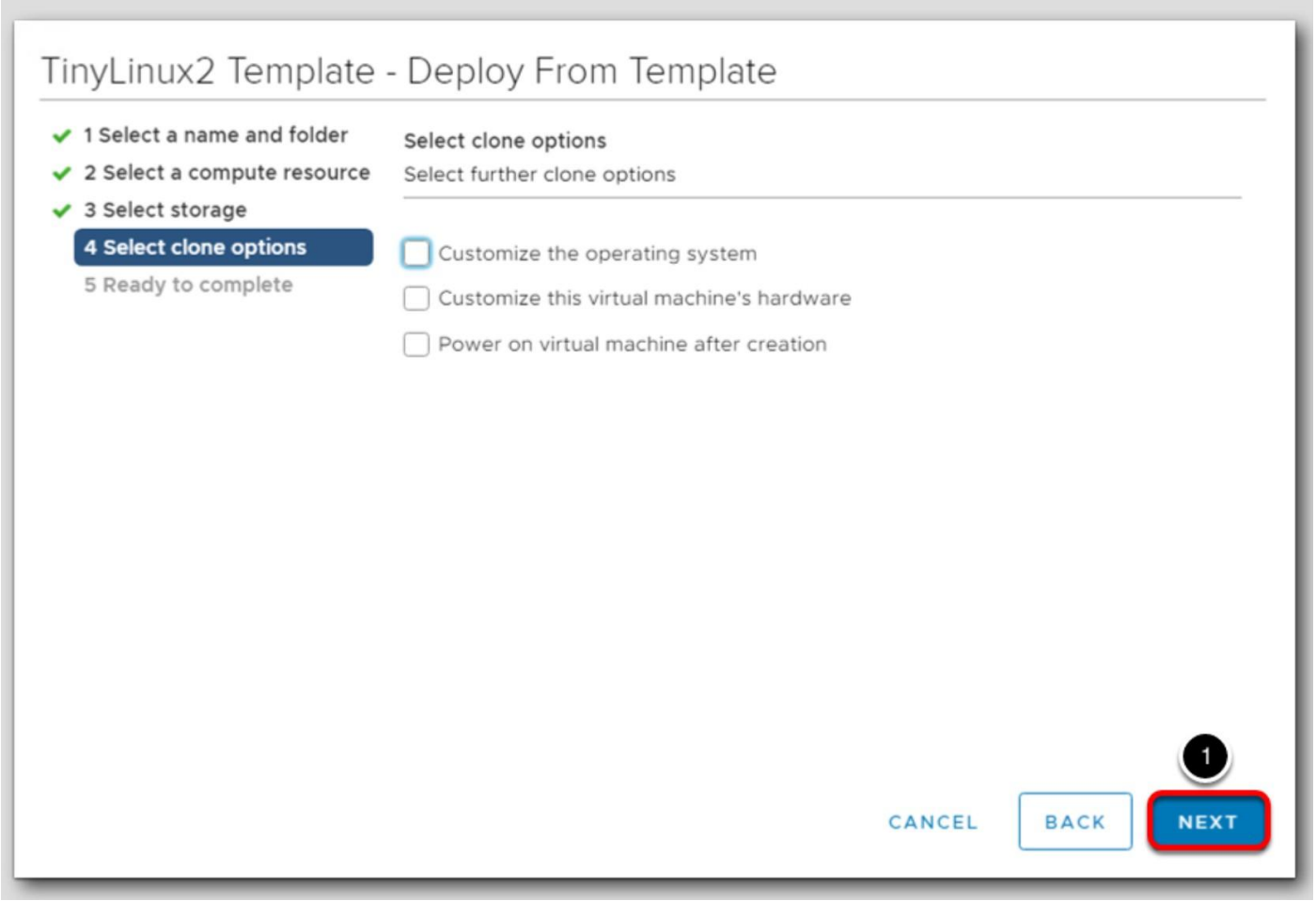
1. Leave the default datastore selected, ds-iscsi01
2. Click Next.



Select clone options

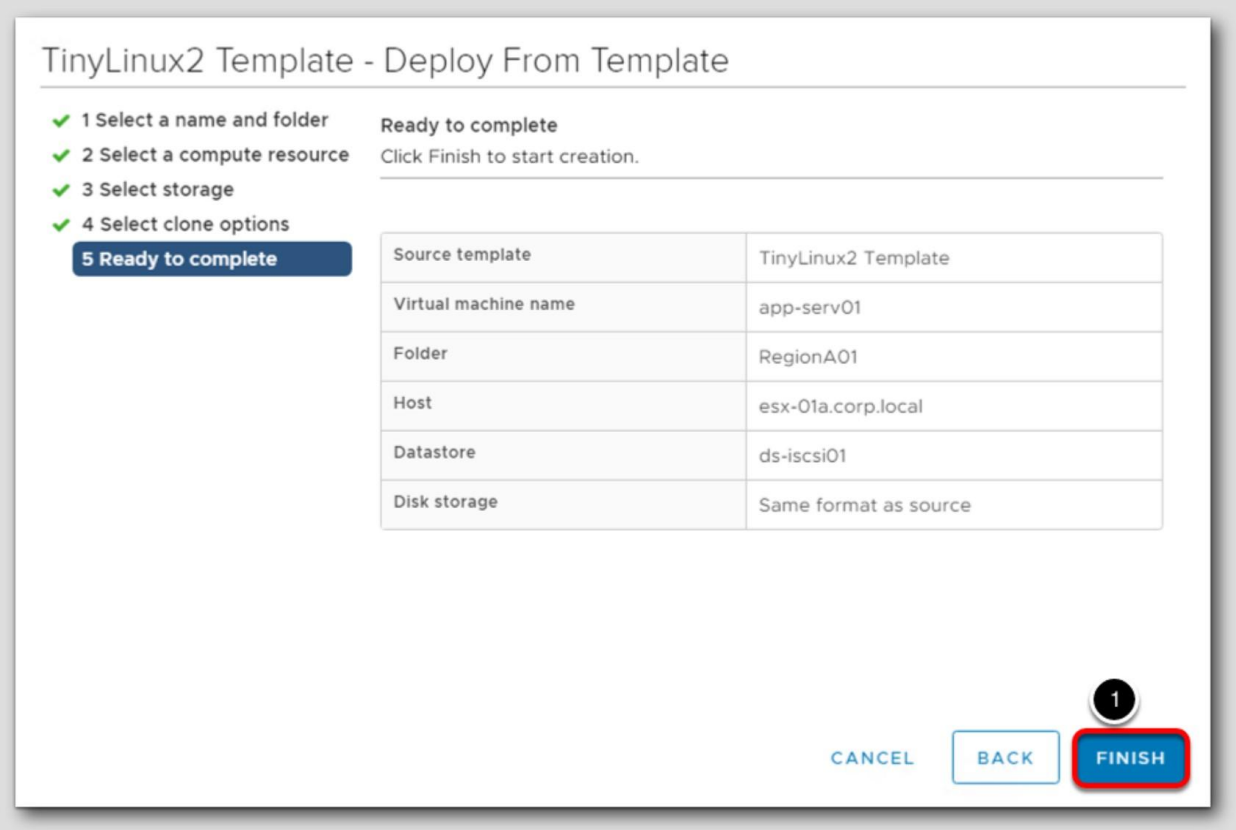
When cloning a virtual machine from a template, the guest operating system and virtual hardware can be modified. For this example, we will not customize the operating system or hardware.

1. Click Next.



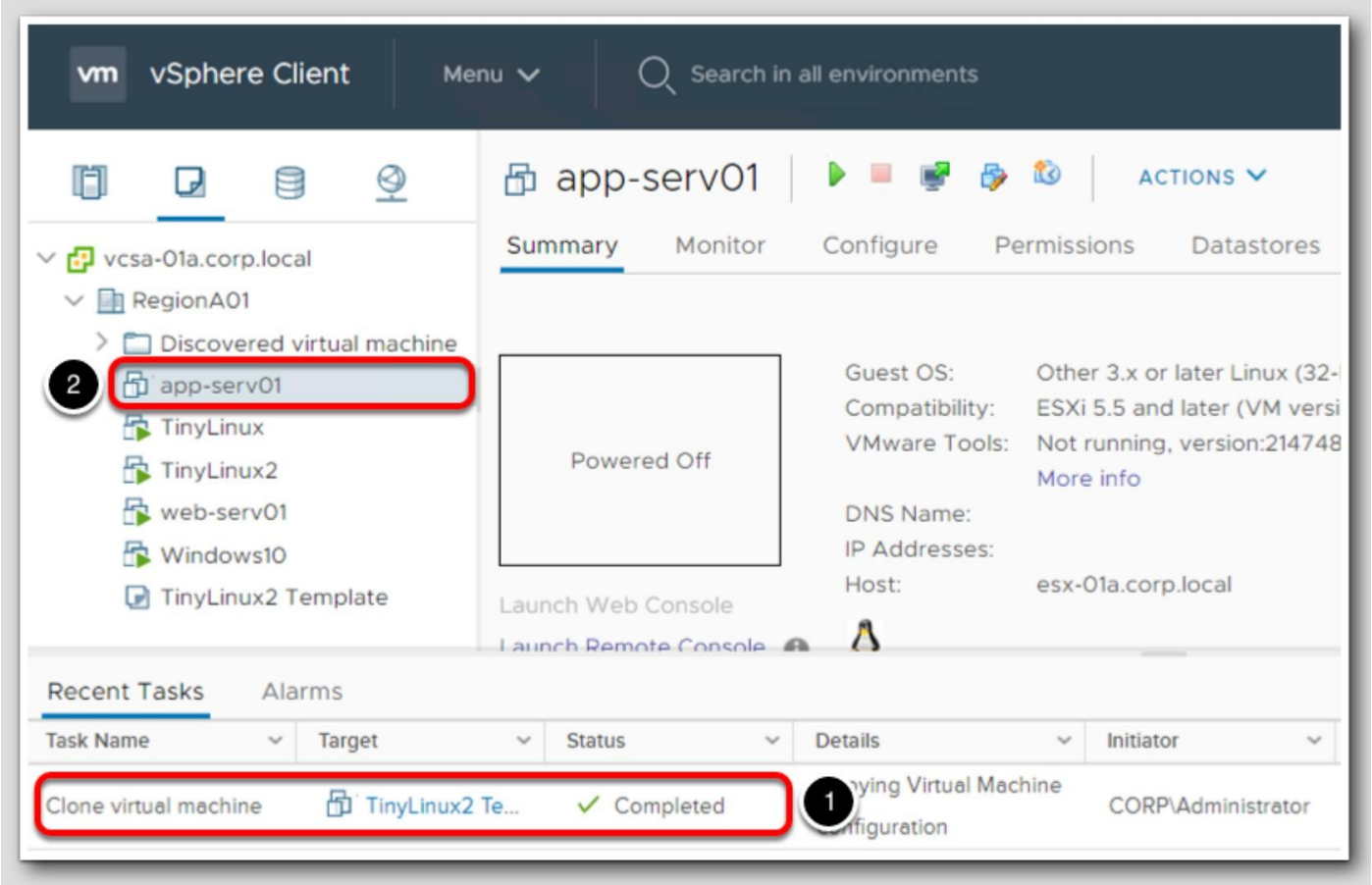
Ready to complete

1. Review the deployment options and then click Finish.



Monitor task progress

1. You can view the Recent Tasks window to monitor the virtual machine being created from the template.
2. When the task is complete, you will see the app-serv01app-serv01 virtual machine in the inventory pane.


vSphere Client
Menu
Search in all environments
app-serv01
ACTIONS
Summary
Monitor
Configure
Permissions
Datastores
vcsa-01a.corp.local
RegionA01
Discovered virtual machine
app-serv01
TinyLinux
TinyLinux2
web-serv01
Windows10
TinyLinux2 Template
2
Powered Off
Guest OS:
Other 3.x or later Linux (32-
Compatibility:
ESXi 5.5 and later (VM versi
VMware Tools:
Not running, version:214748
More info
DNS Name:
IP Addresses:
Host:
esx-01a.corp.local
Launch Web Console
Launch Remote Console
Recent Tasks
Alarms
Task Name
Target
Status
Details
Initiator
Clone virtual machine
TinyLinux2 Te...
Completed
1
CORP\Administrator

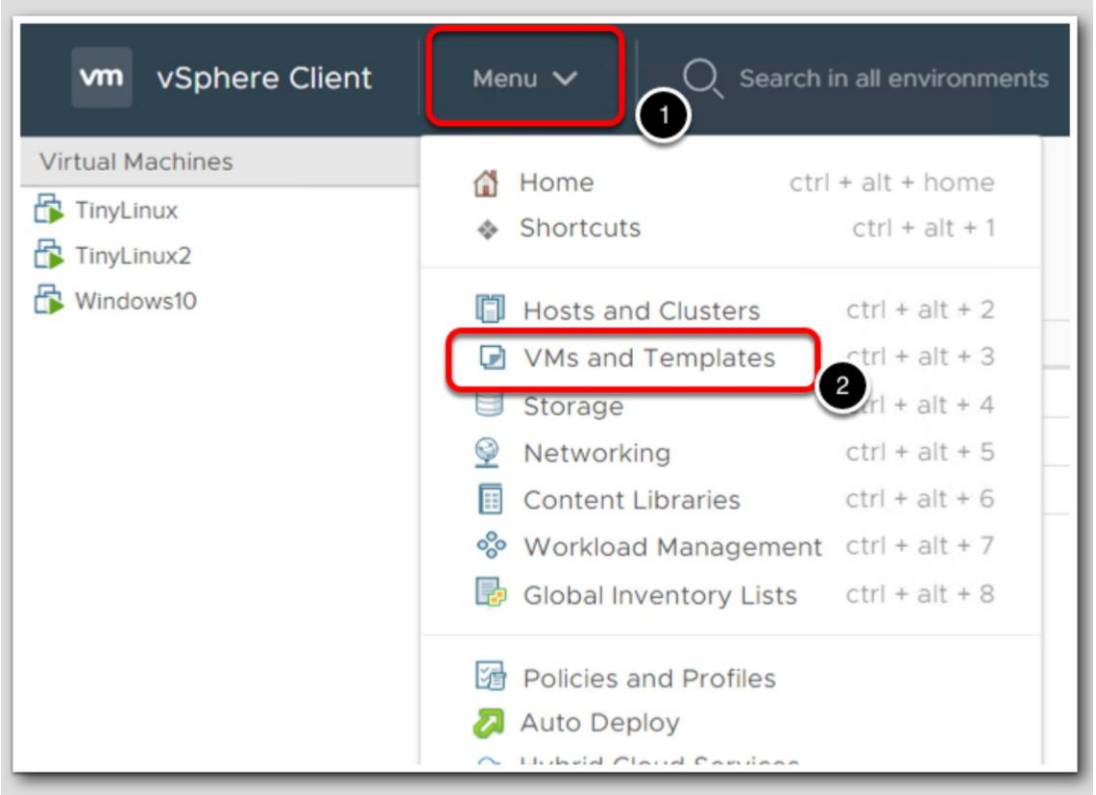
Practical 7: Create a content library to clone and deploy virtual machines.

Create a Virtual Machine

In the next steps, we will create a virtual machine and then, install an operating system.

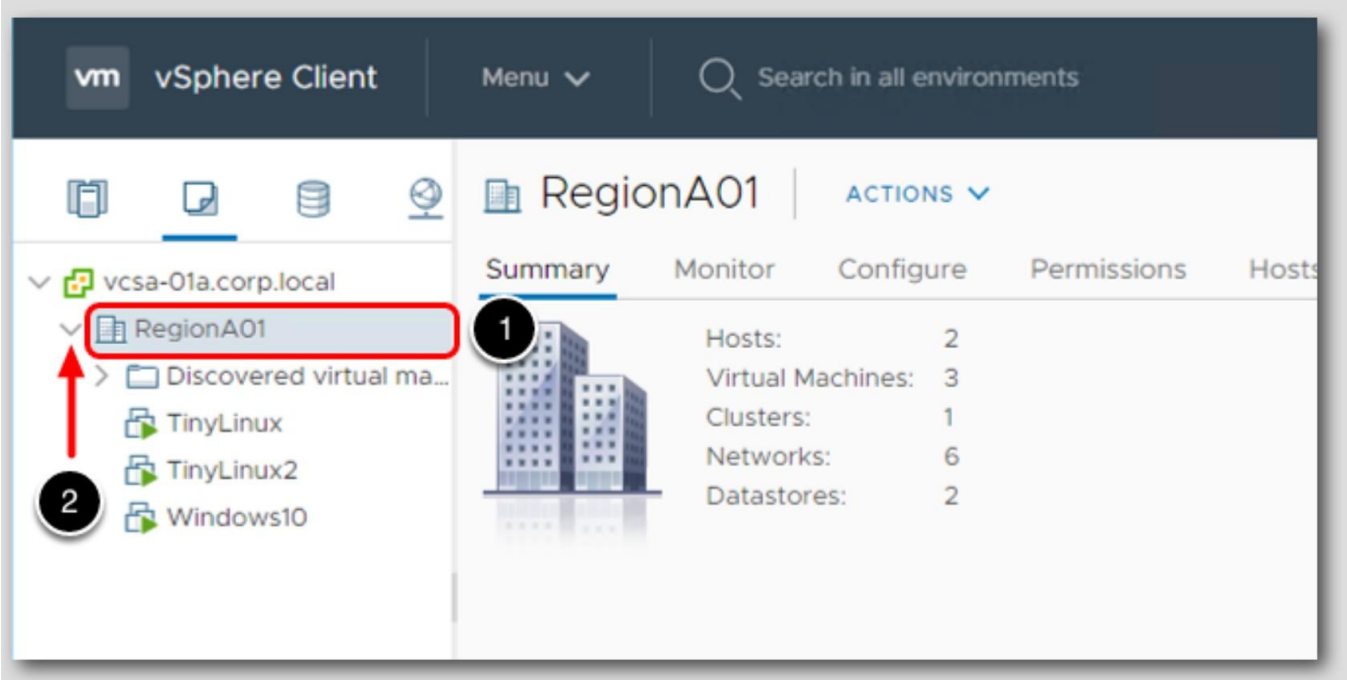
1. To return to the VMs and Templates view, click on Menu.

2.Select VMs and Templates.



Select and Expand Datacenter

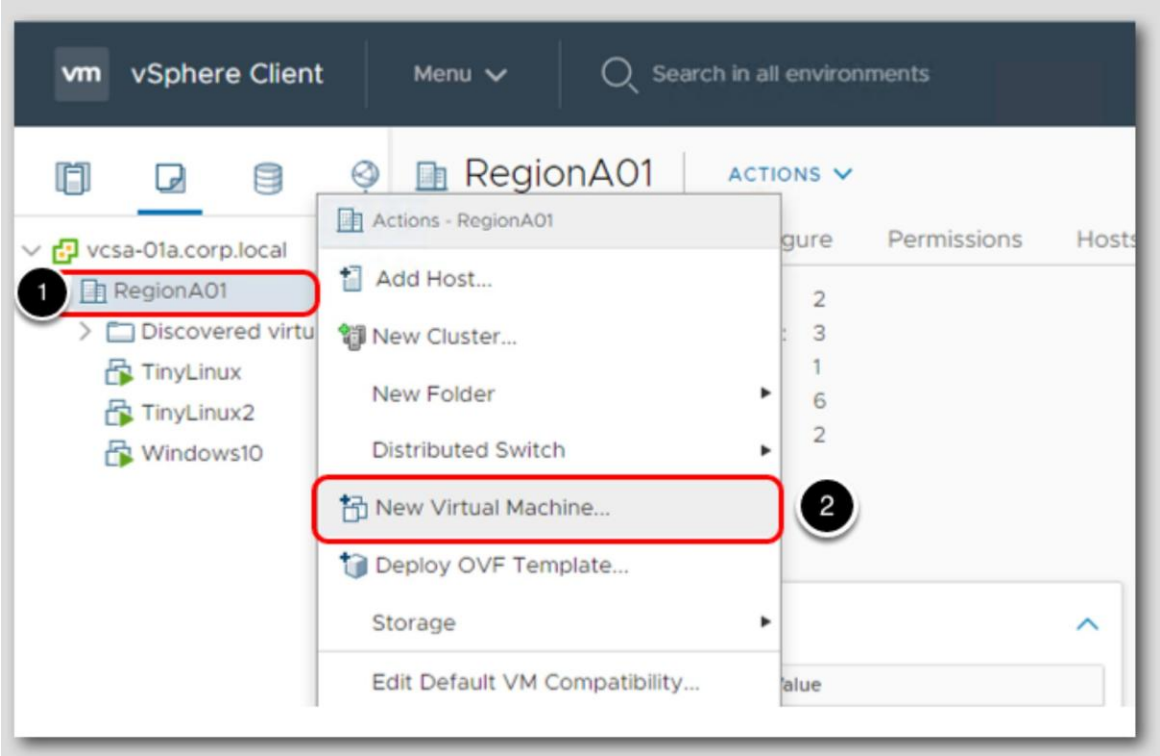
1. Click on RegionA01 Datacenter.
2. Expand RegionA01 Datacenter so the virtual machines under it can be seen.



Start the New Virtual Machine Wizard

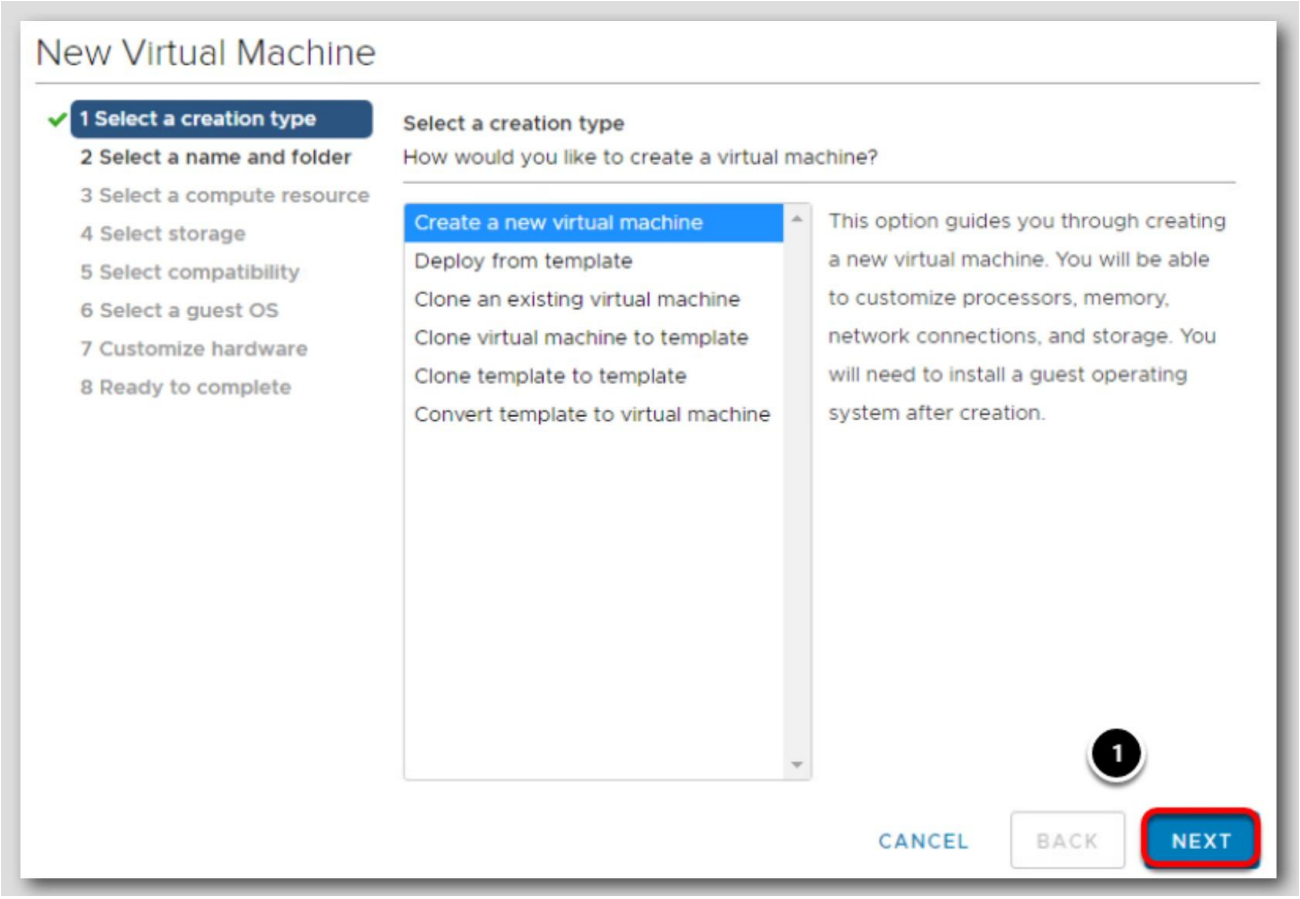
1. Right-click on RegionA01 Datacenter.
2. Click New Virtual Machine to start the new virtual machine wizard.

This wizard is used to create a new Virtual Machine and place it in the vSphere inventory.



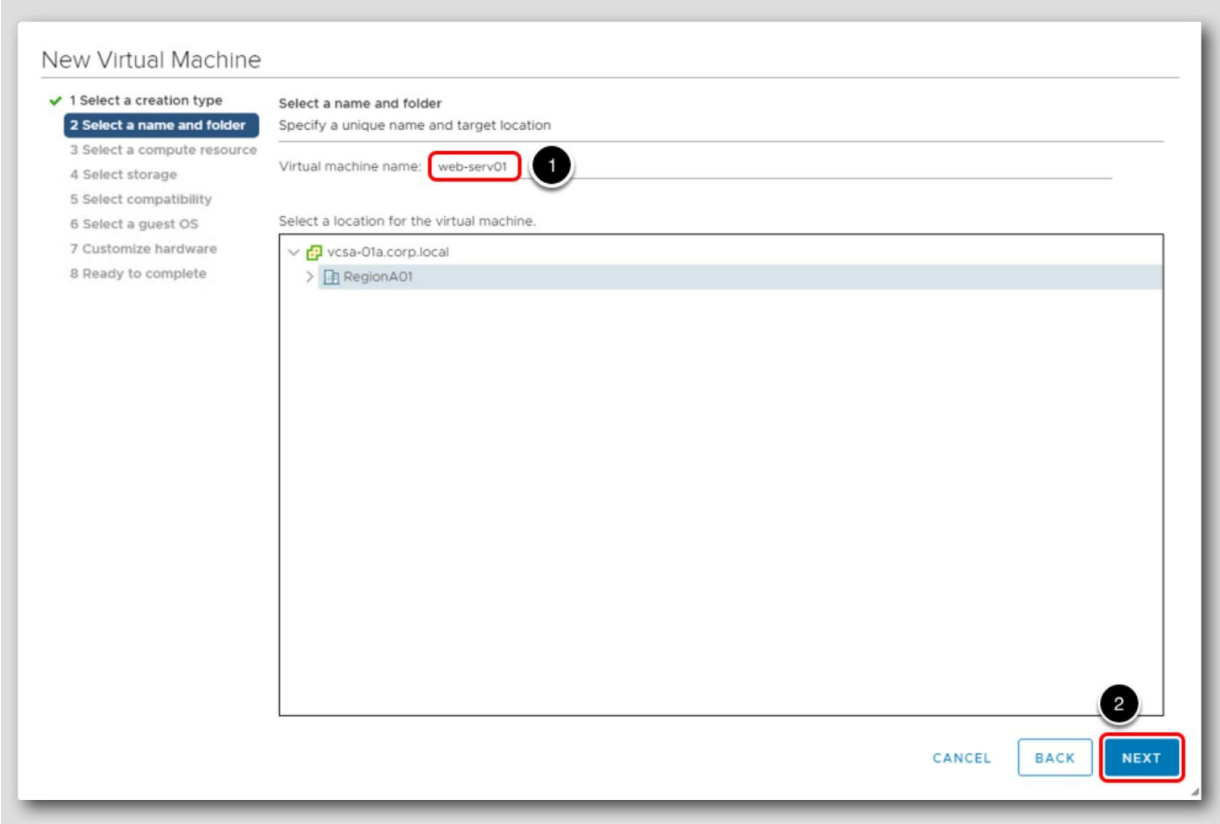
Virtual Machine wizard

Since the Create a new virtual machine wizard is highlighted, just click Next.



Name the Virtual Machine

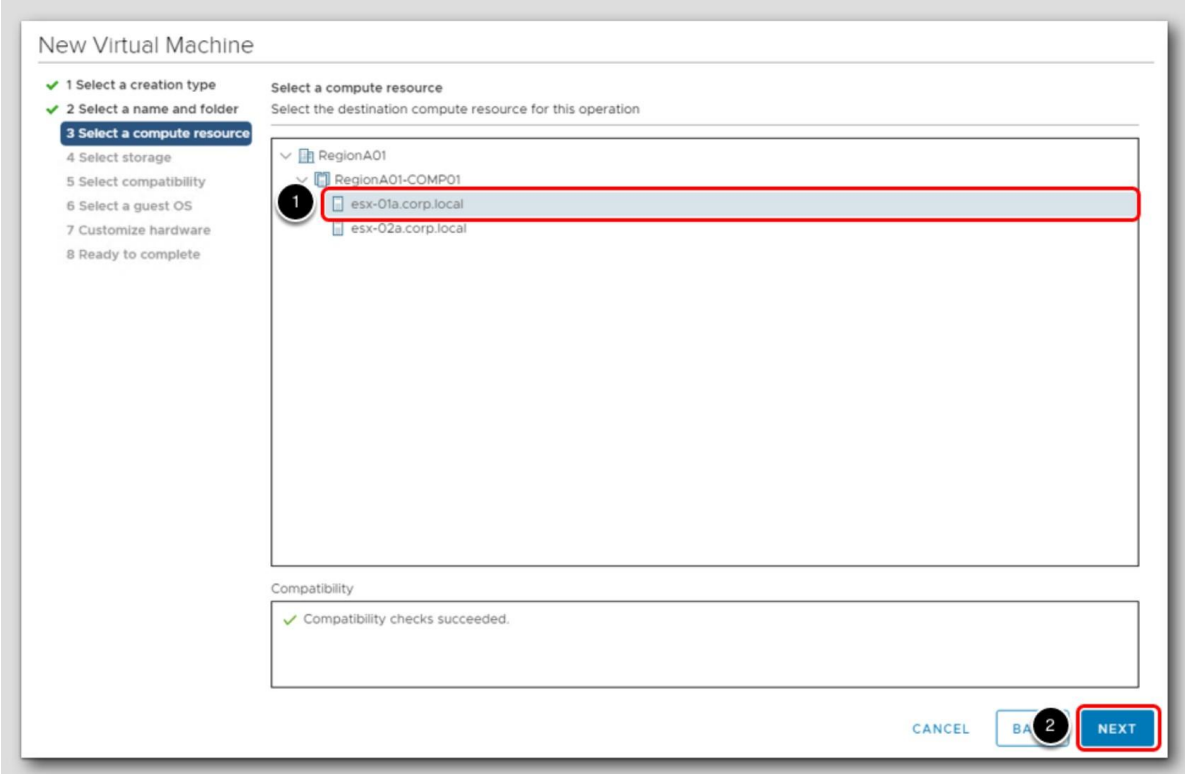
1. Enter web-serv01 for the name of the new virtual machine.
2. Click Next.



Virtual Machine Placement

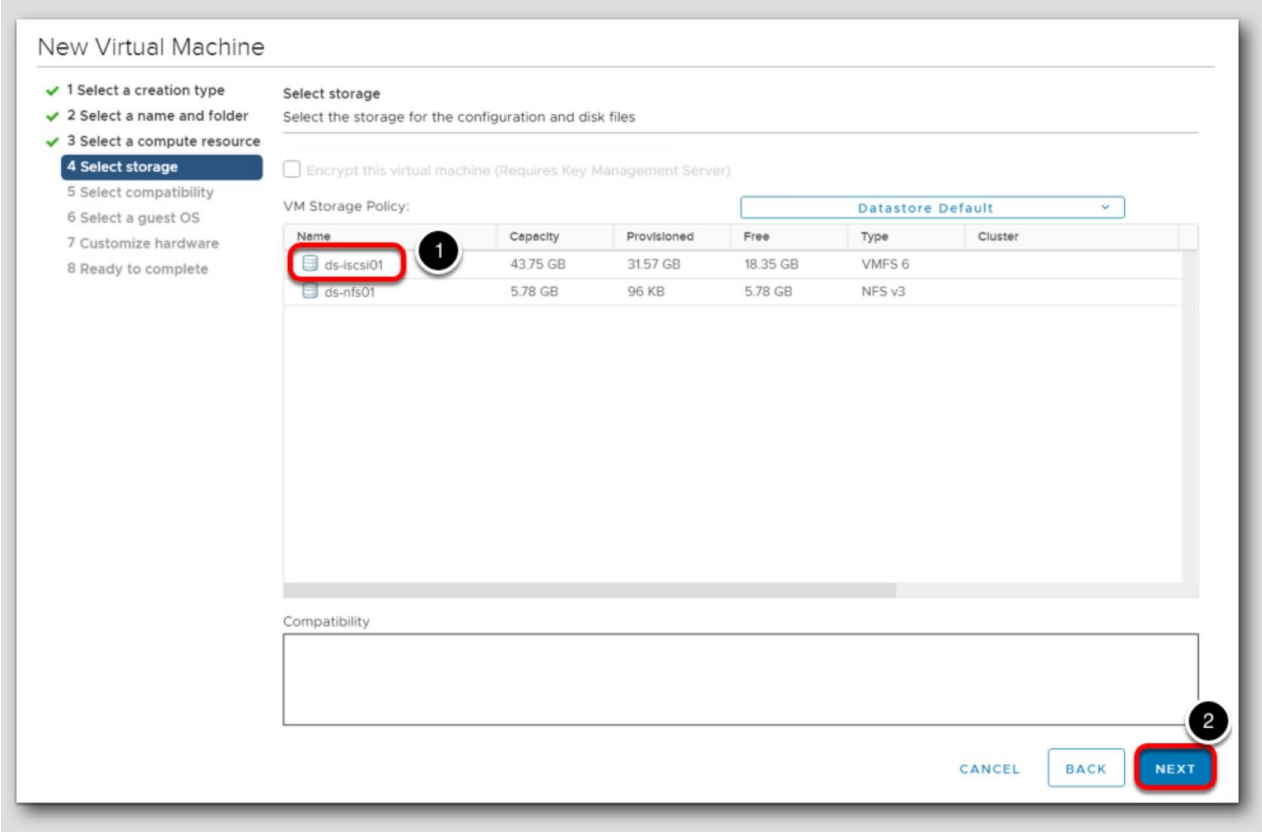
Because Distributed Resource Scheduler (DRS) is not enabled, you just have to select a host to use for the VM. More details on DRS will be covered later in this module.

1. Click esx-01a.corp.local.
2. Click Next.



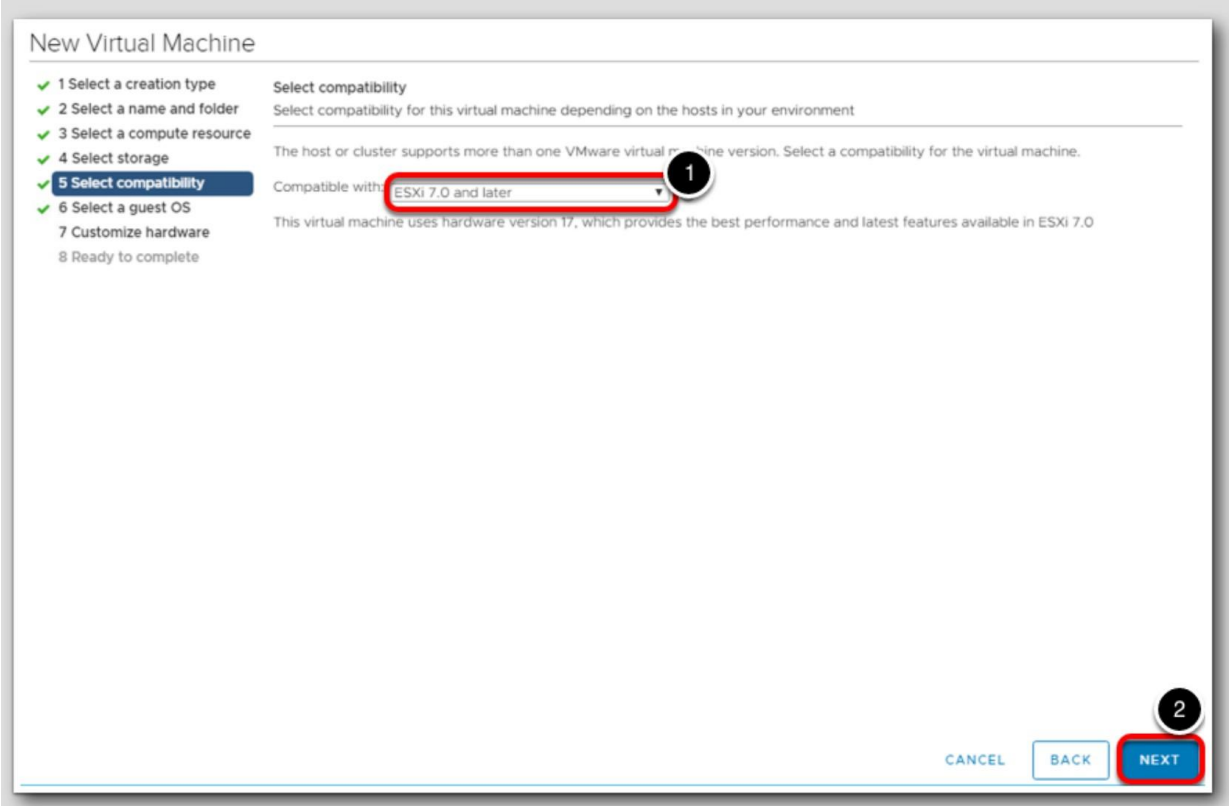
Select Storage

1. Ensure the ds-iscsi01 datastore is selected.
2. Click Next.



Compatibility 1. Select ESXi 7.0 and later.

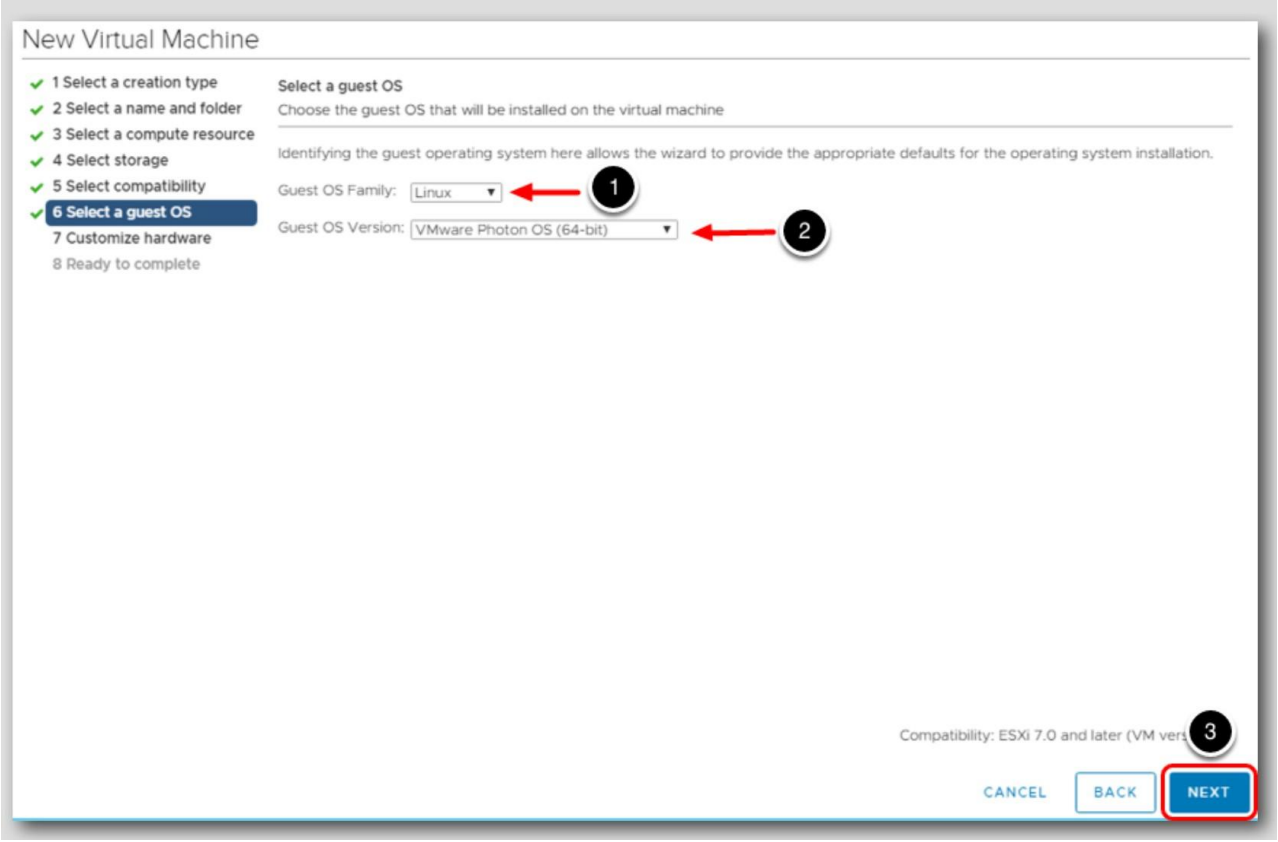
2. Click Next to accept.



Guest OS

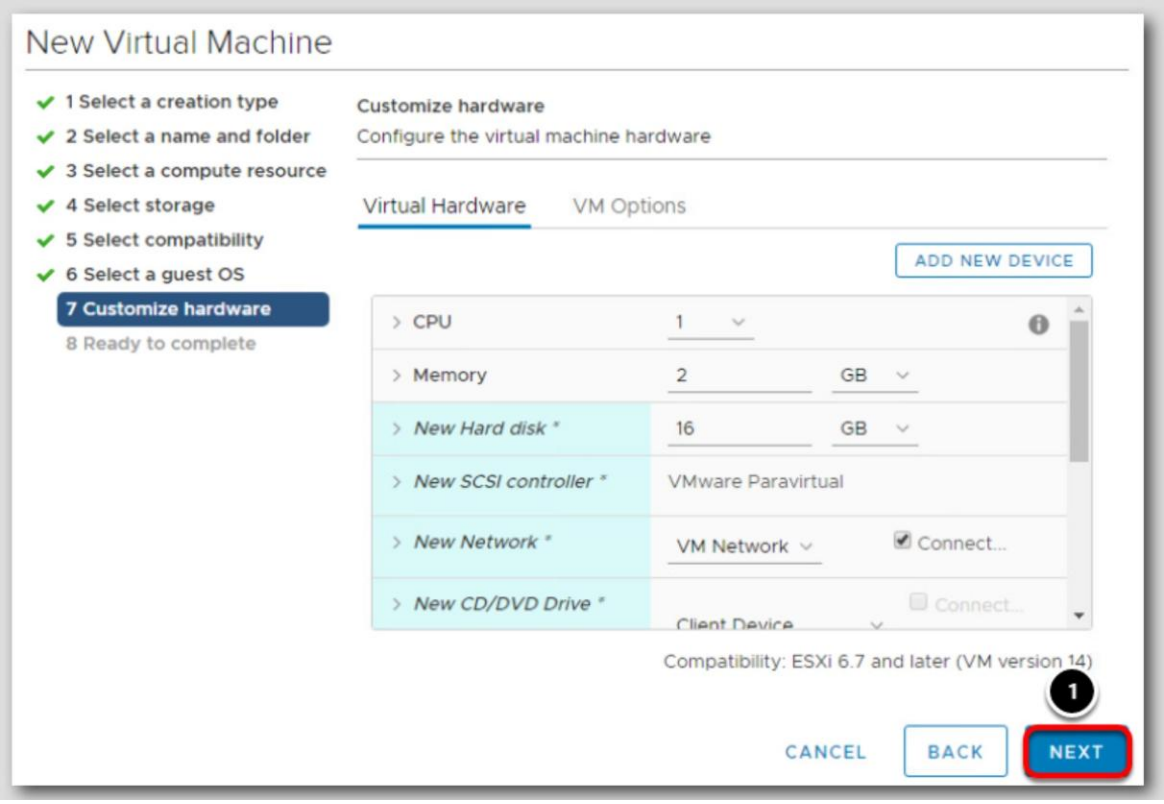
In this step, we will be selecting what operating system we will be installing. When we select the operating system, the supported virtual hardware and recommended configuration is used to create the virtual machine. Keep in mind this does not create a virtual machine with the operating system installed, but rather creates a virtual machine that is tuned appropriately for the operating system you have selected.

1. For the Guest OS Family, select LinuxLinux from the drop-down menu.
2. For the Guest OS Version, select VMware Photon OS (64-bit).
3. Click Next to continue.



Change Virtual Disk Size

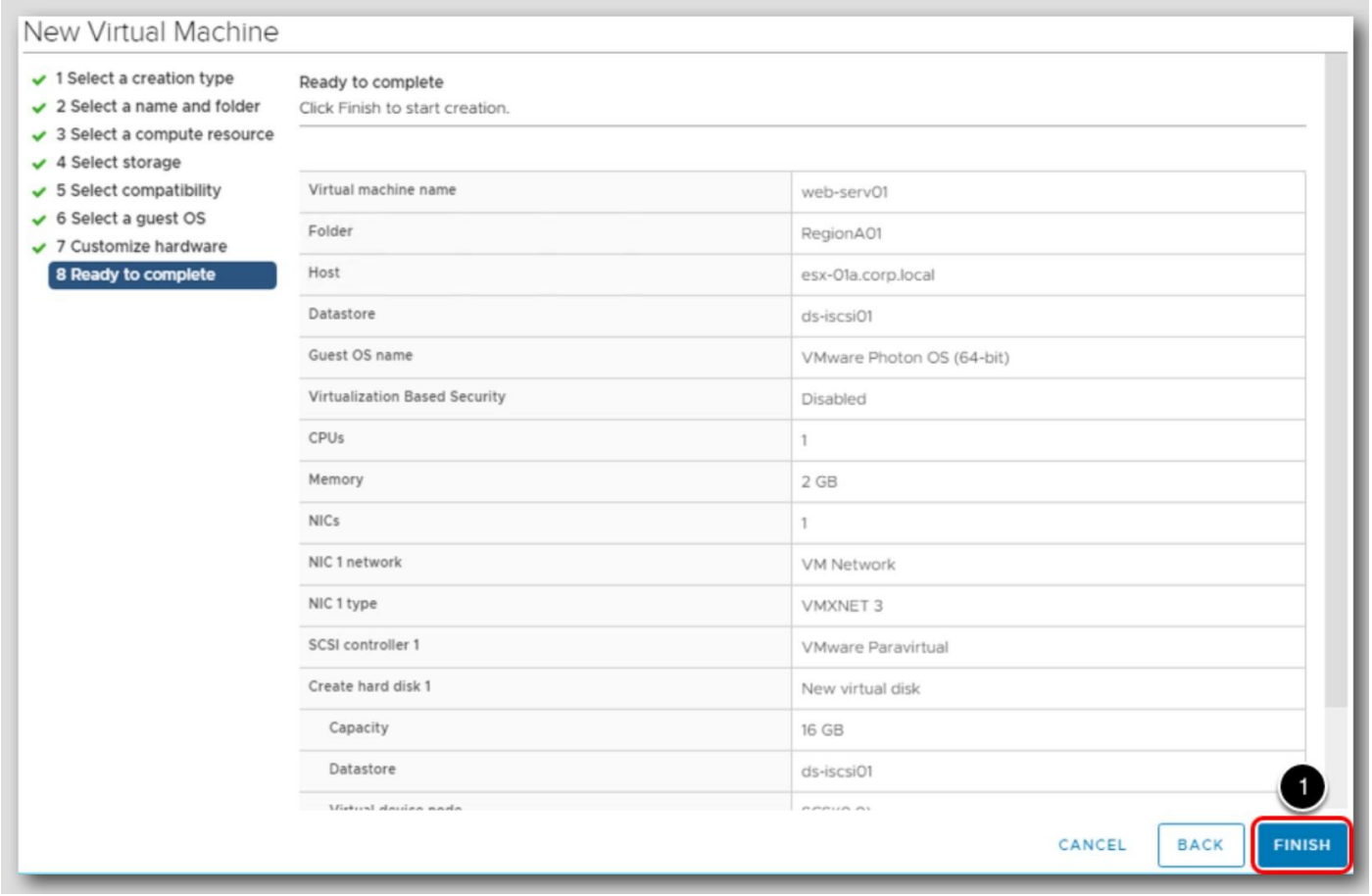
1. Leave the default settings and click Next



Ready to complete

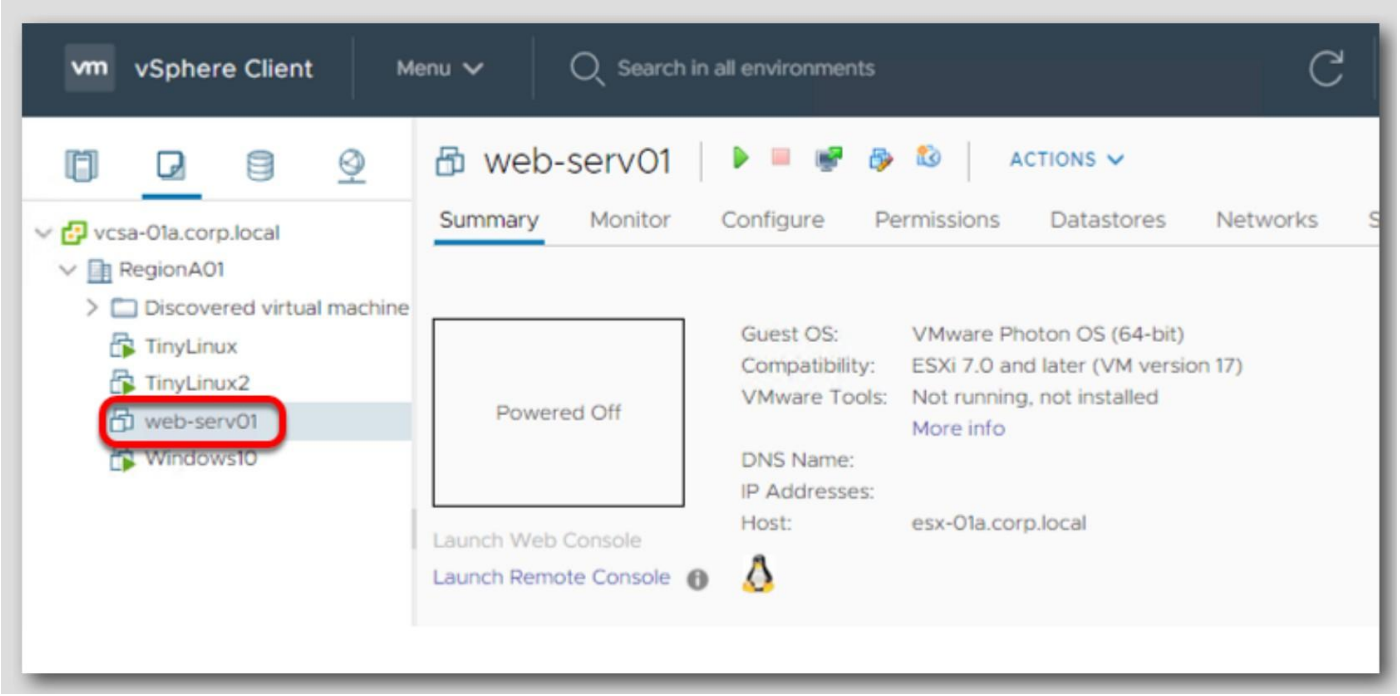
The settings for the virtual machine can be verified prior to it being created.

1. Click Finish to create the virtual machine.



Newly created virtual machine

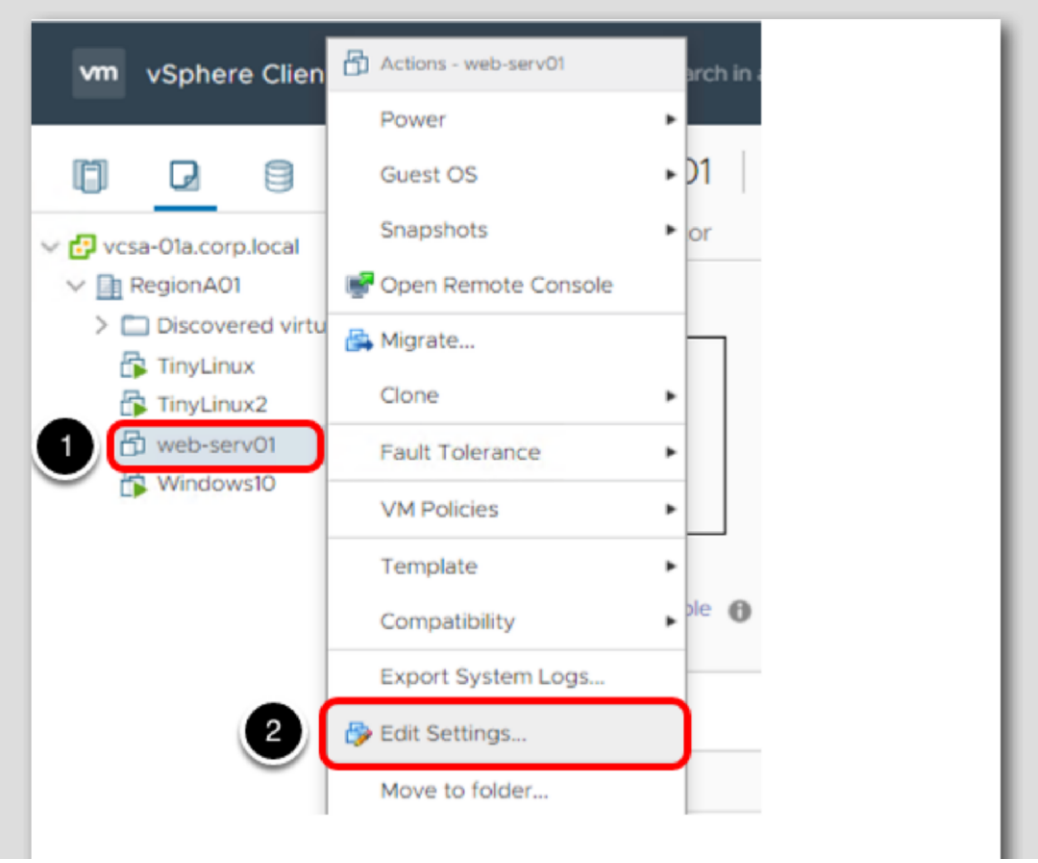
Congratulations on creating your first virtual machine web-serv01.



Attaching an ISO to a Virtual Machine

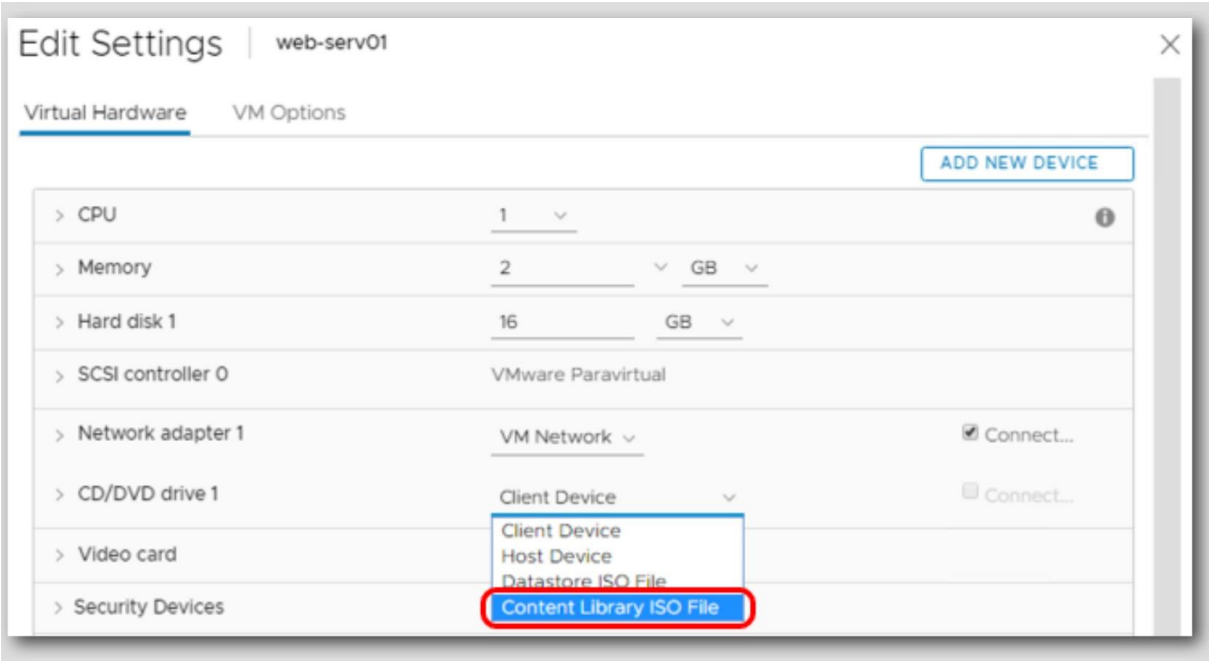
To make it easier to install operating systems on virtual machines, ISO images can be used. These can be kept in the same storage used for virtual machines. In addition, vCenter offers a Content Library as a repository. Content Libraries can then be synchronized to ensure every location is using the same versions.

1. To attach an ISO image to the virtual machine we just created, make sure web-serv01 is selected.
2. Right-click on web-serv01 and select Edit Settings…



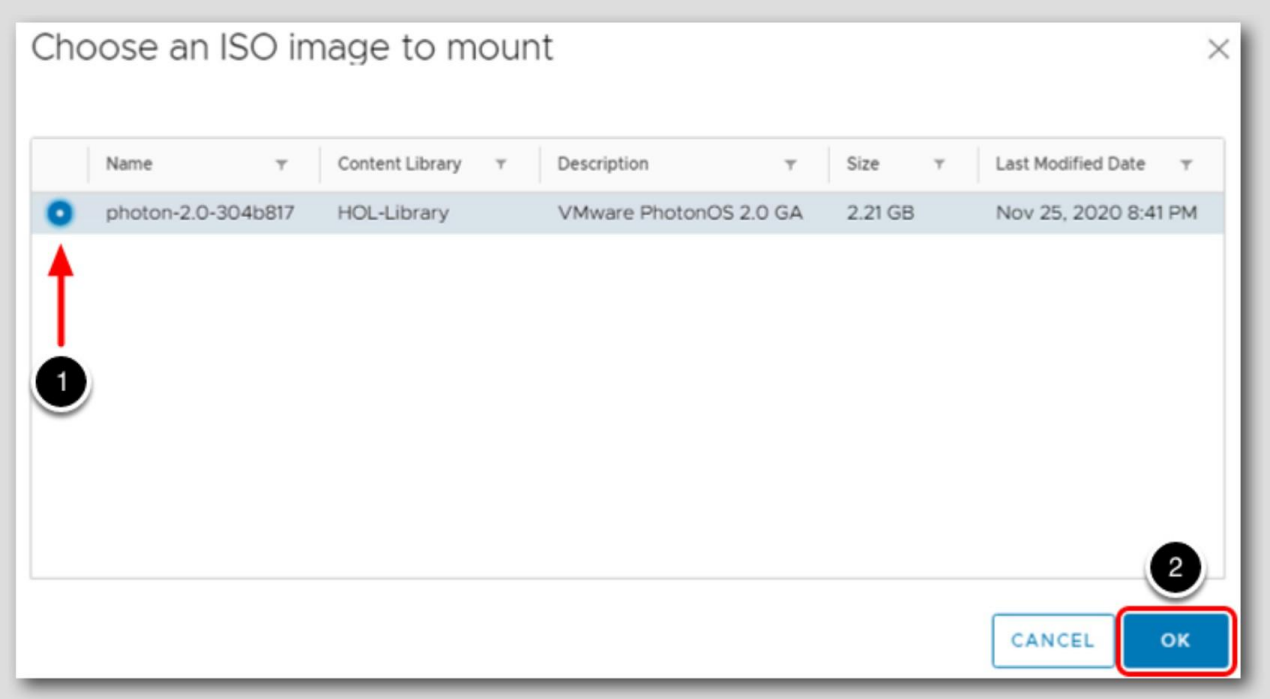
Content Library ISO File

1. From the CD/DVD drive 1 drop-down menu, select Content Library ISO file.



Select Photon

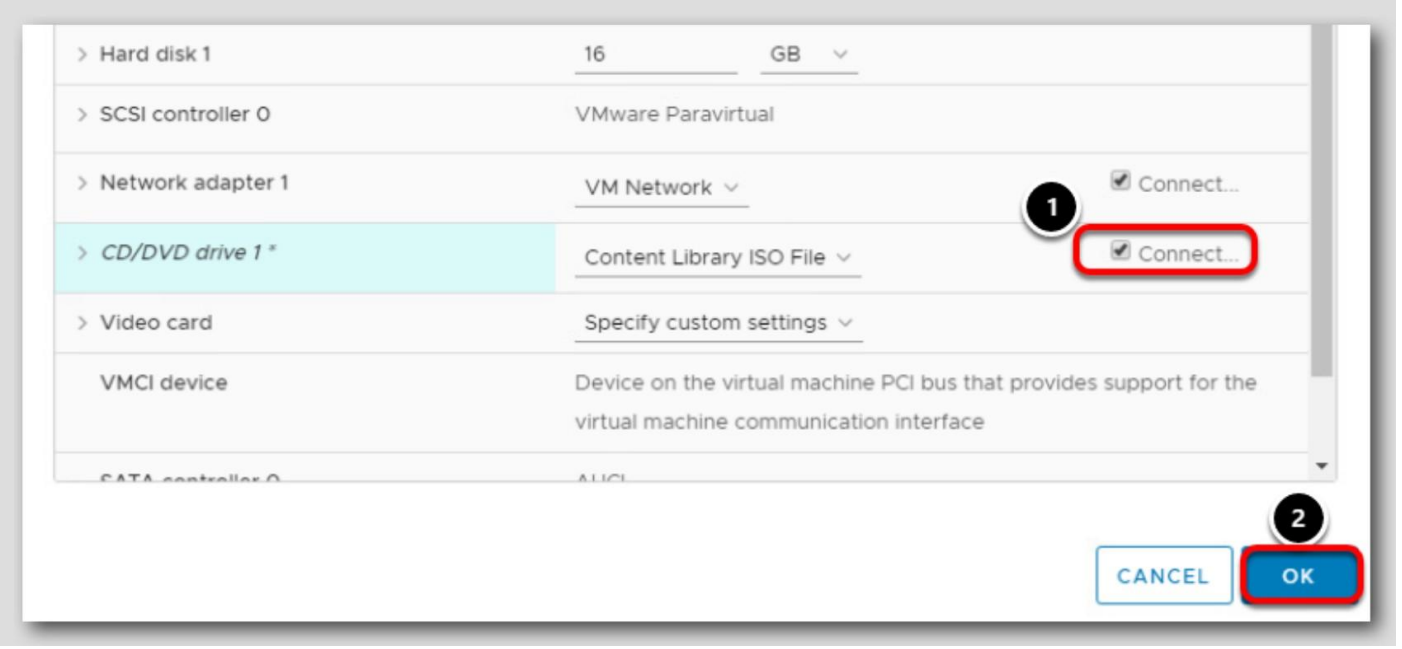
1. Click the radio button next to photon-2.0-304b817.
2. Click OK.



Connect the drive

Finally, we want to attach or connect the ISO image to the virtual machine.

1. Click the Connected check box next to CD/DVD drive 1CD/DVD drive 1.
2. Click OK

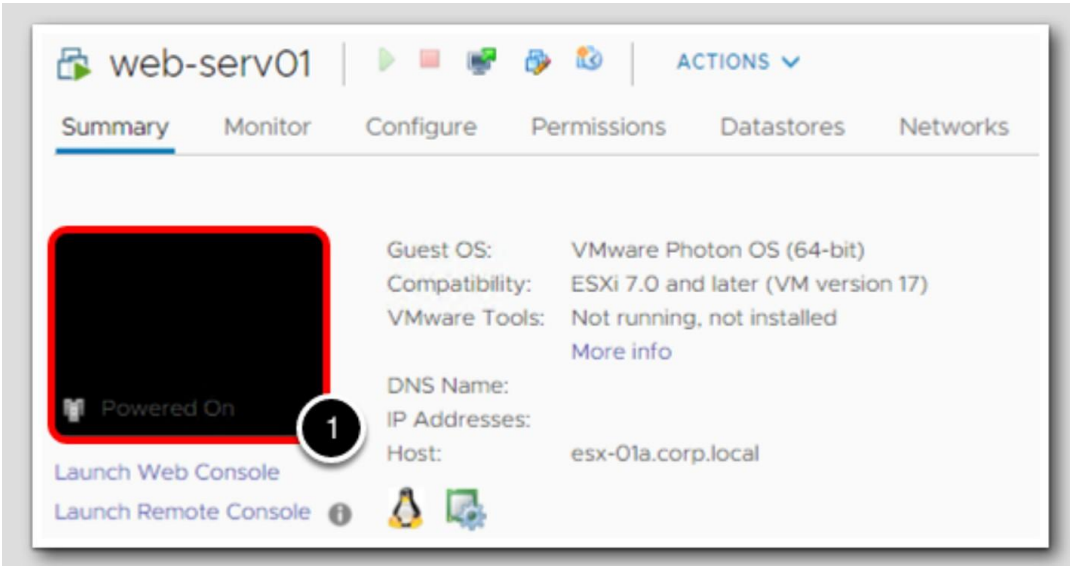


Power on web-serv01

1. Click the green green play button to power on the virtual machine and start the installation.

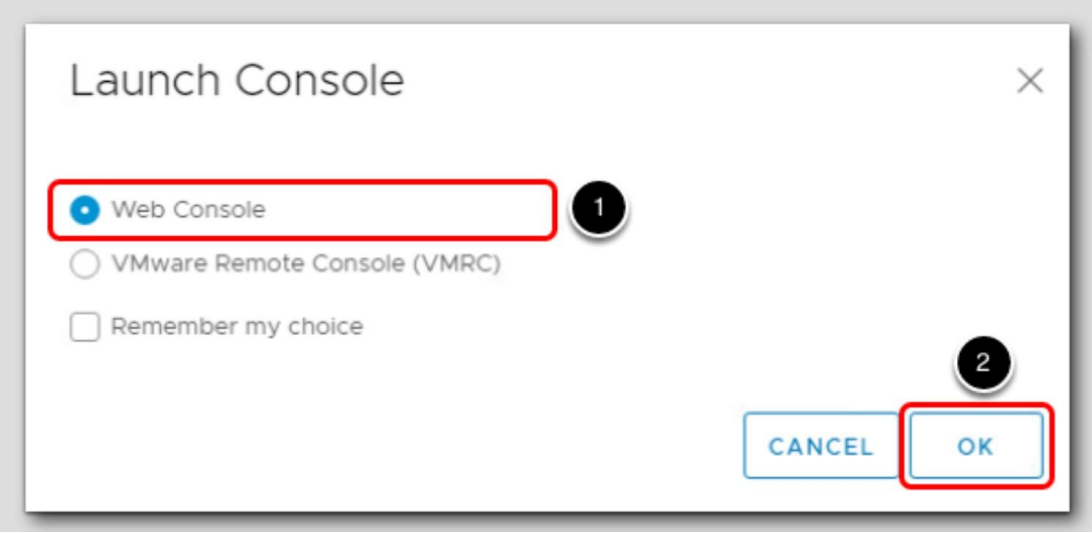
Launch Console

1. To launch the console window, click anywhere in the console window screen.



Web Console

1. Select the Web Console.
2. Click OK.



Note you also have the option of using the VMware Remote Console (VMRC). This is console is a separate application that needs to be installed on your local device as opposed to the Web Console which will launch in a new browser tab. The VMRC can be useful in certain situations when you need more capabilities, like attaching devices or power cycling options.

Photon Boot Screen

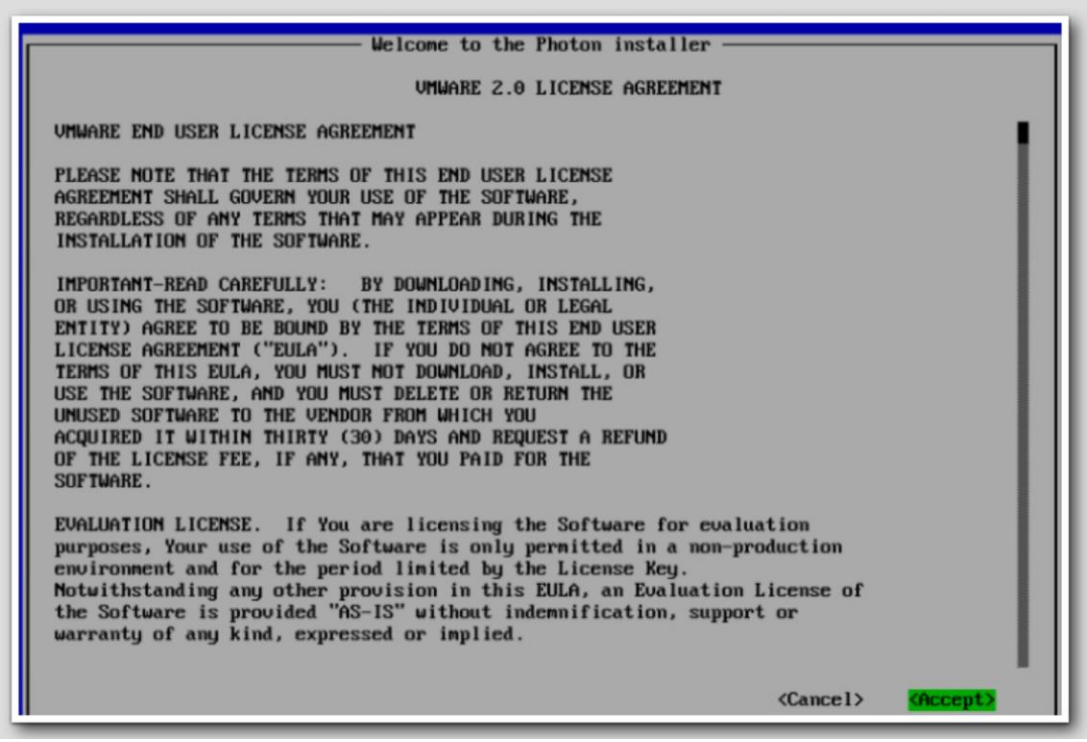
A new tab will open and you will be presented with the Photon OS boot screen.

1. Press the Enter key to start the installation process.

License Agreement

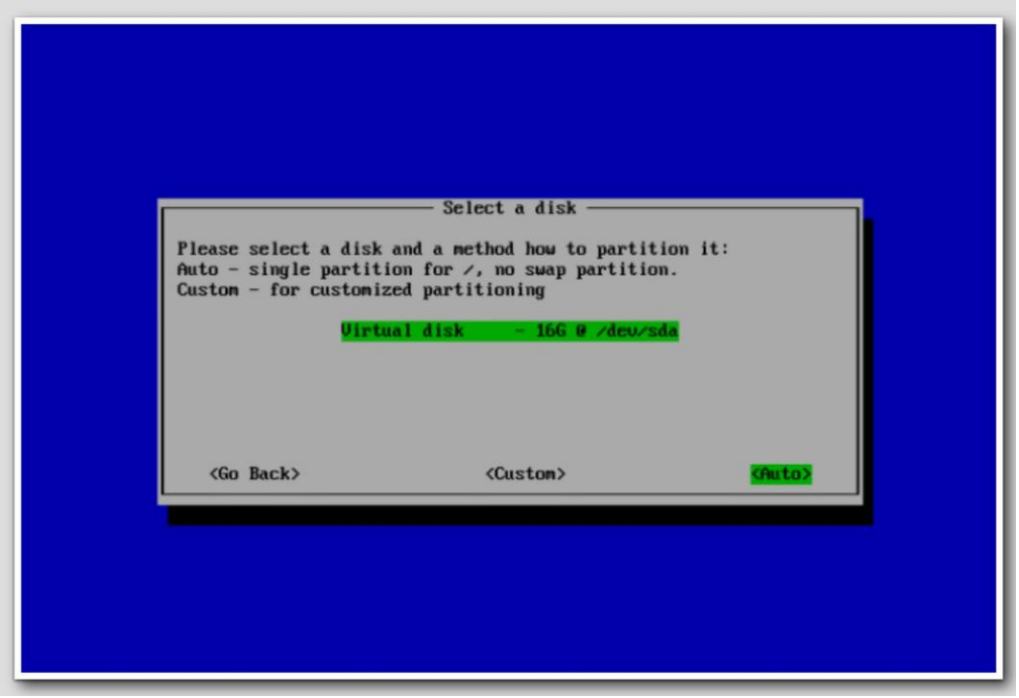
After the boot process is complete, you will be presented with a license agreement.

1. Press Enter to accept.



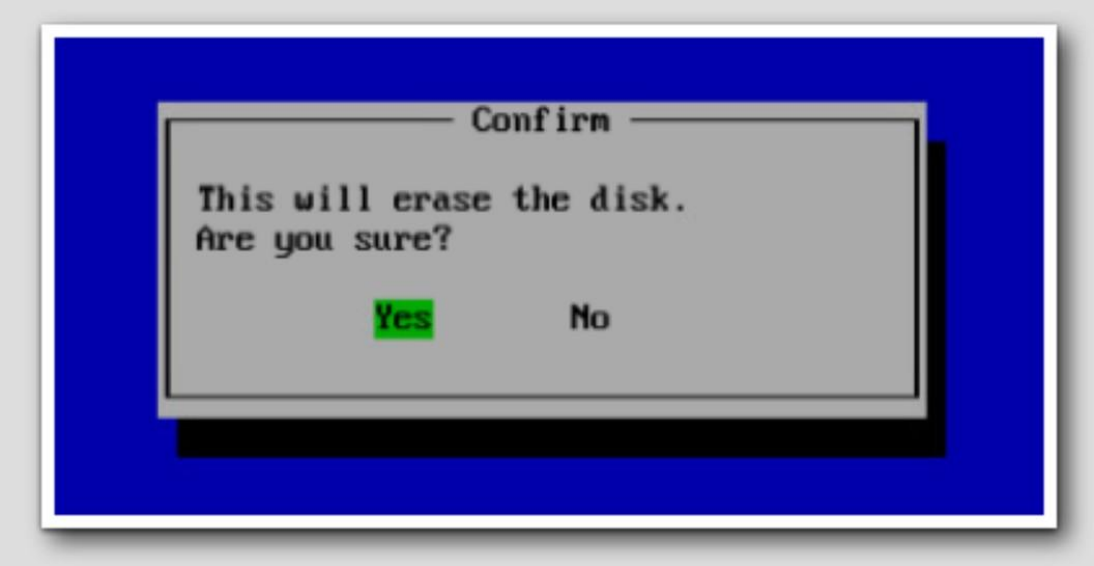
Select Disk

1. Press Enter to accept the selected disk and use the auto partitioning option



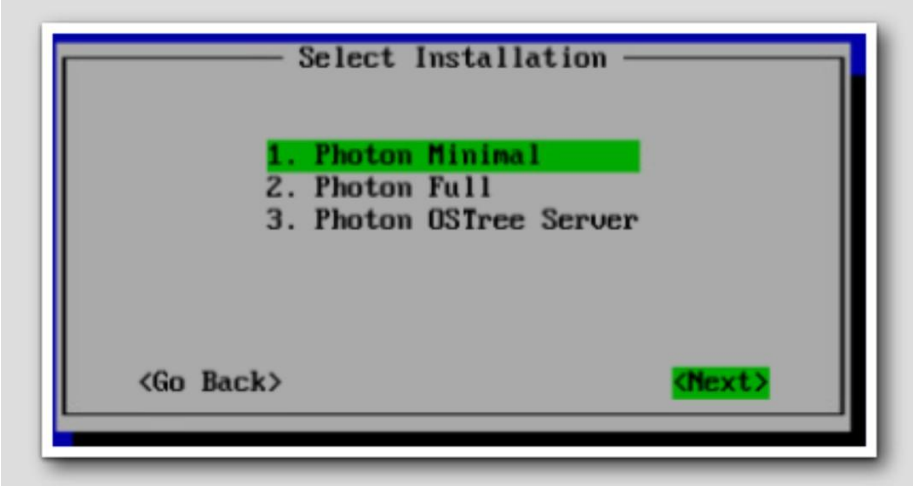
Confirm

1. Press Enter confirm the disk should be erased.



Select Installation

1. At the Select Installation screen, make sure the default option of 1. Photon Minimal is selected.
2. Press the Enter key.

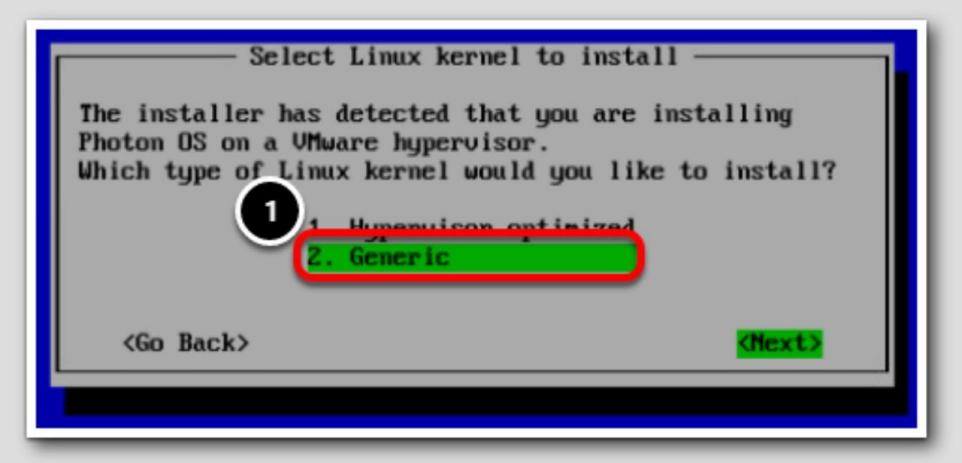


Linux Kernel

1. Use the arrow key to select 2. Generic.

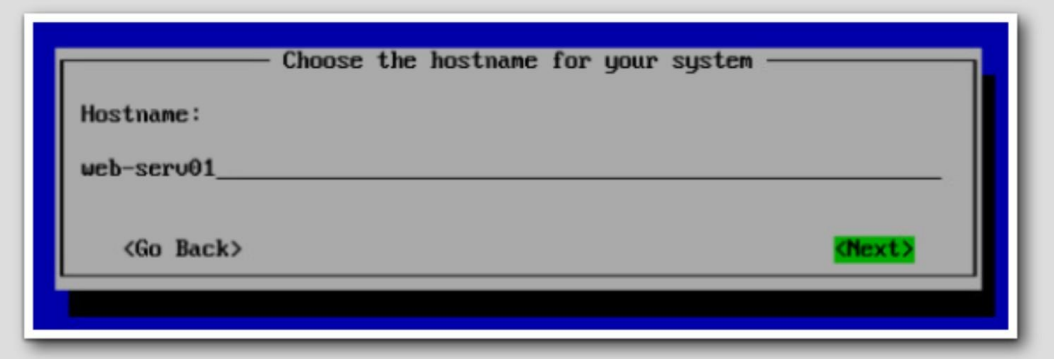
2.Press the Enter key.

NOTE: If 1. Hypervisor optimized is selected, the virtual machine will not boot. This is due to the unique environment the Hands-on Labs are running in.



Rename Host

1. Use the Backspace key to remove the default hostname.
2. Type web-serv01.
3. Press the Enter key.



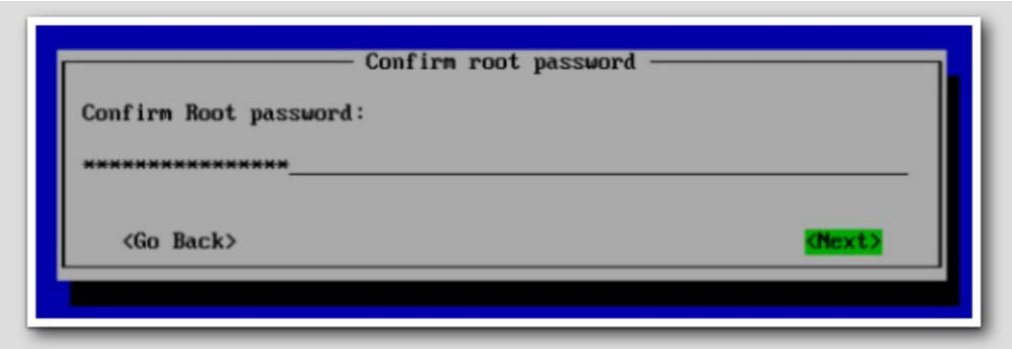
Password

1. For the password, use VMware1!VMware1!

Note that Photon requires a complex, non-dictionary password, which is why the typical password is being repeated.

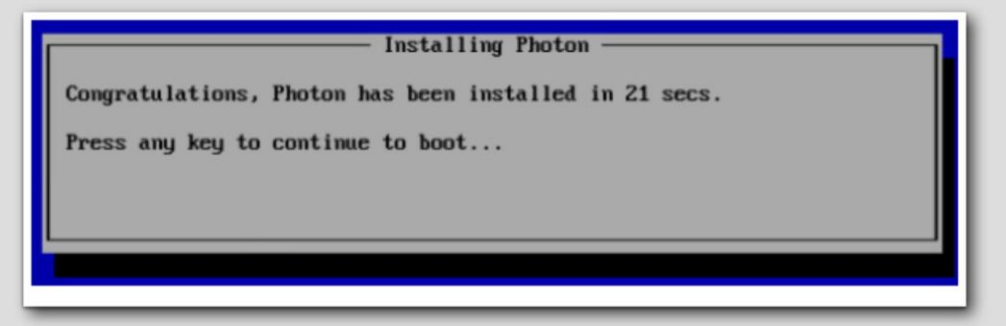
Confirm Password

1. Type VMware1!VMware1! again to confirm the password.
2. Press the Enter key.



Installation Complete

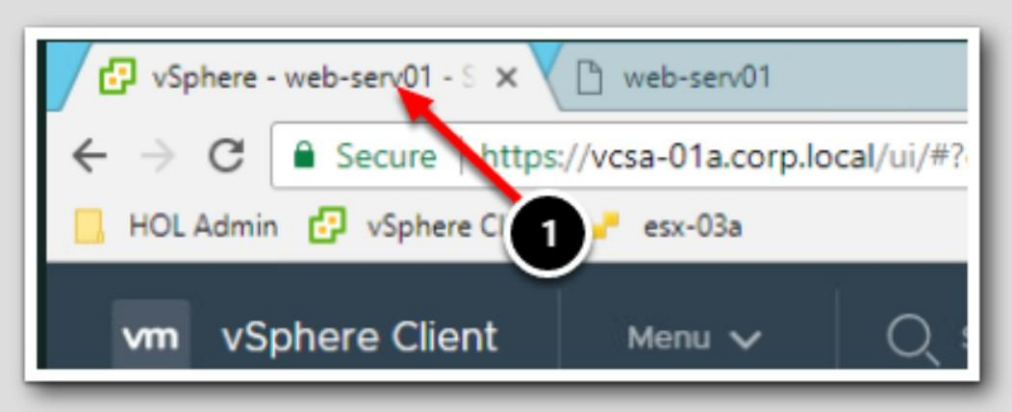
After a minute or two, the installation will be complete.
Press a key to reboot the virtual machine. After a minute or two, the system should boot the login prompt.



vSphere Tab

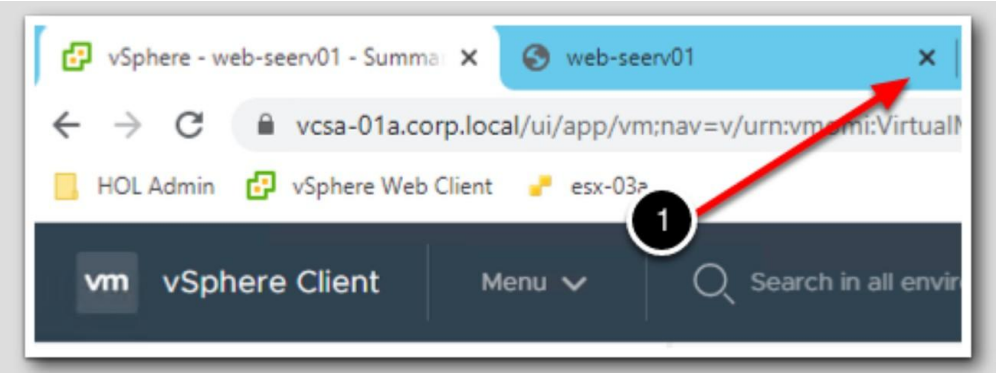
Now that the operating system has been installed and is up and running, the ISO image needs to be disconnected from the virtual machine.

1. Select the vSphere- web-serv01 tab.



web-serv01 Console

1. Click the 'X' to close the console window for web-serv01



Practical 8: Use vSphere vMotion and vSphere Storage vMotion to migrate virtual machines.

Planned downtime typically accounts for over 80% of datacenter downtime. Hardware maintenance,server migration, and firmware updates all require downtime for physical servers. To minimize the impact of this downtime, organizations are forced to delay maintenance until inconvenient and difficult-to-schedule downtime windows.

The vMotion functionality in vSphere makes it possible for organizations to reduce planned downtime because workloads in a VMware environment can be dynamically moved to different physical servers without service interruption. Administrators can perform faster and completely transparent maintenance operations, without being forced to schedule inconvenient maintenance windows. With vSphere vMotion, organizations can:

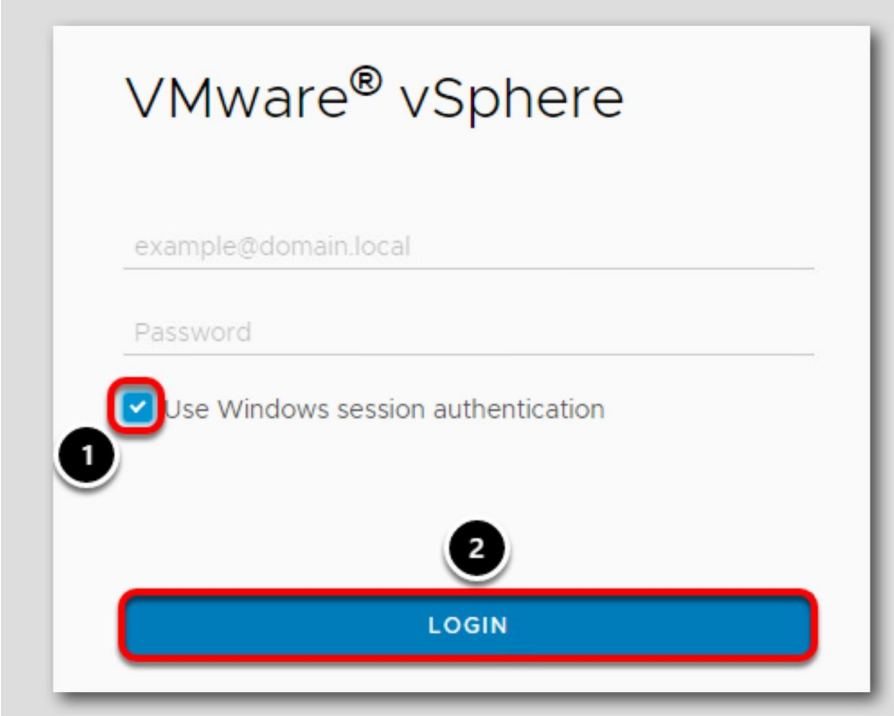
- Eliminate downtime for common maintenance operations.
- Eliminate planned maintenance windows.
- Perform maintenance at any time without disrupting users and services.

Launch Google Chrome web browser

1.Click on the Chrome Icon on the Windows Quick Launch Task Bar.

Enter credentials and log in

1. Select "Use Windows session authentication" check box.
2. Select Login.



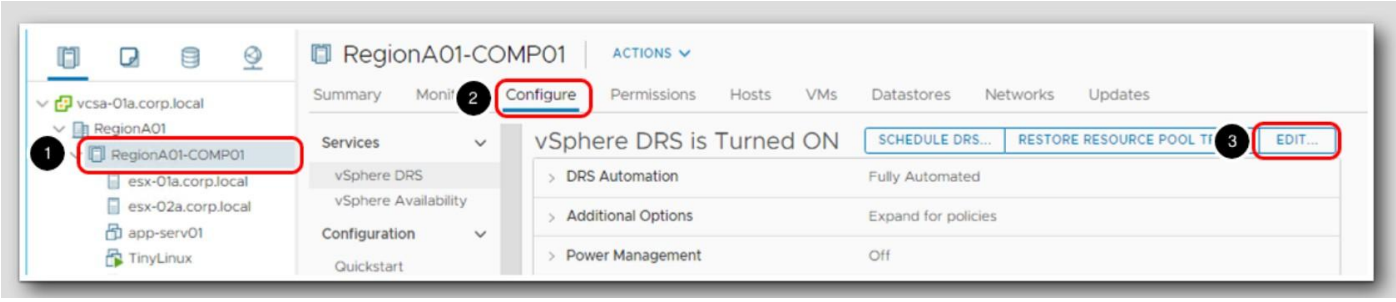
If credentials aren't saved, use the following:

- username: administrator@corp.local ● password: VMware1!

Edit Cluster Settings

We will disable DRS and then migrate all of the virtual machines esx-02a.corp.local hosts over to esx-01a.corp.local. This will also help prepare us for the next lesson on Performance.

1. Select RegionA01-COMP01
2. Click the Configure tab
3. Click the Edit button

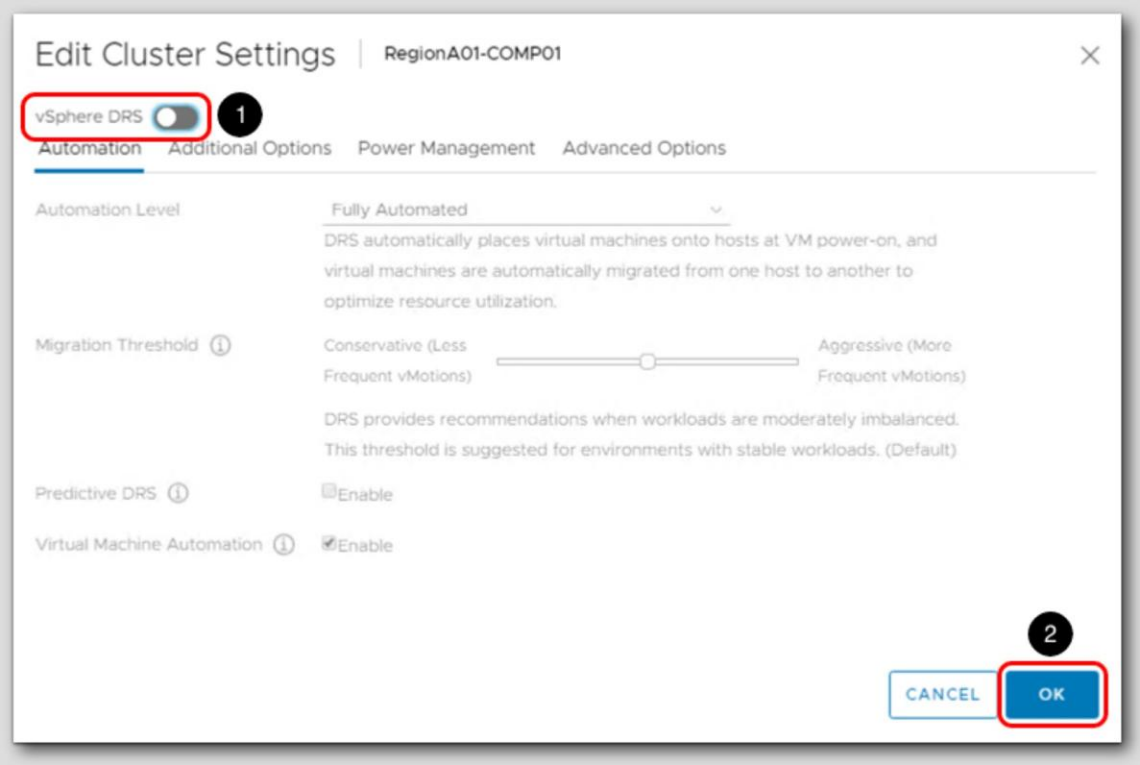


Disable DRS

1. Flip the switch to disable vSphere DRS.

2.Click OK

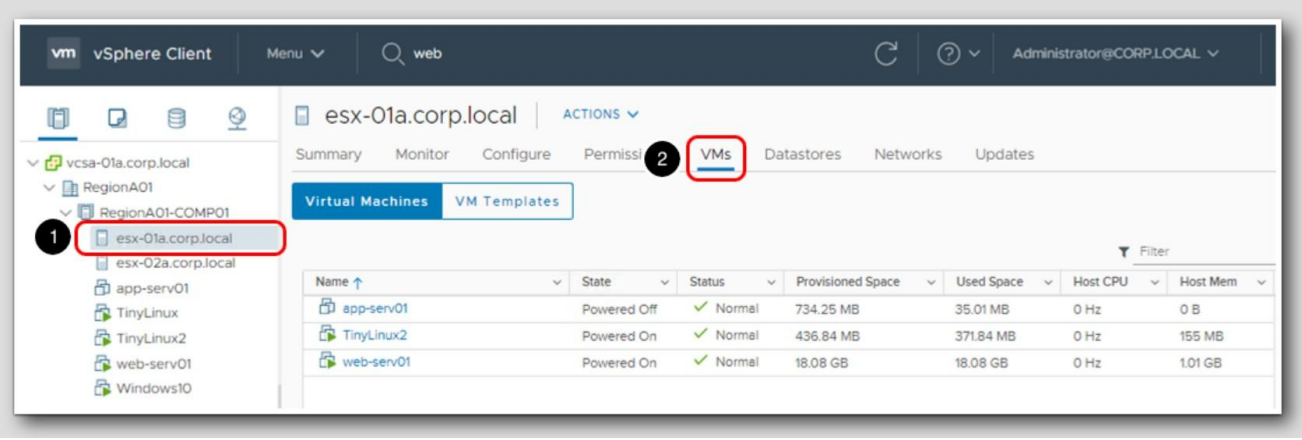
By disabling DRS, this will prevent the virtual machines from being migrated back to esx-01a.corp.local.



Migrating to esx-02a.corp.local

1. Select esx-01a.corp.local
2. Click the VMs tab

Depending on what other modules you have taken, you may see more VMs

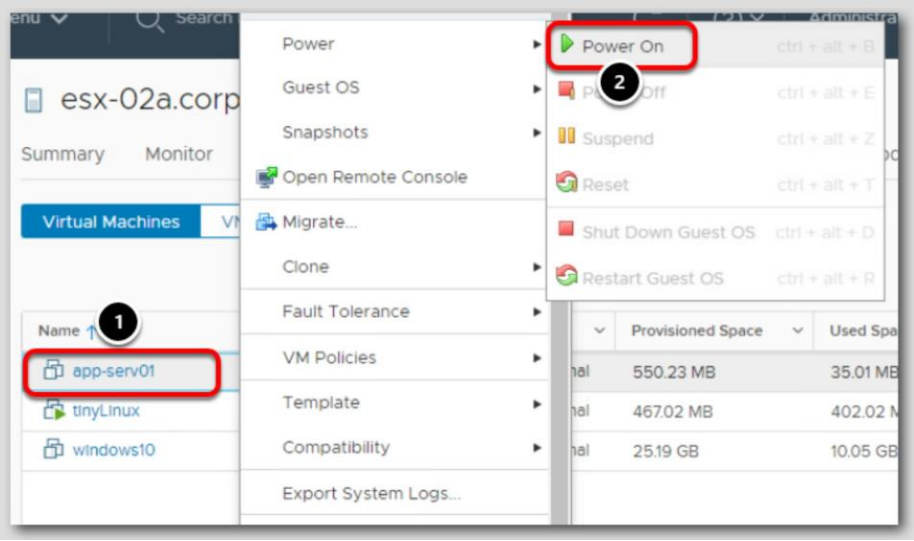


Power on VMs

1. Look for any virtual machines that are Powered Off and select them. Multiple virtual machines can be selected by holding the Ctrl key and clicking on them.

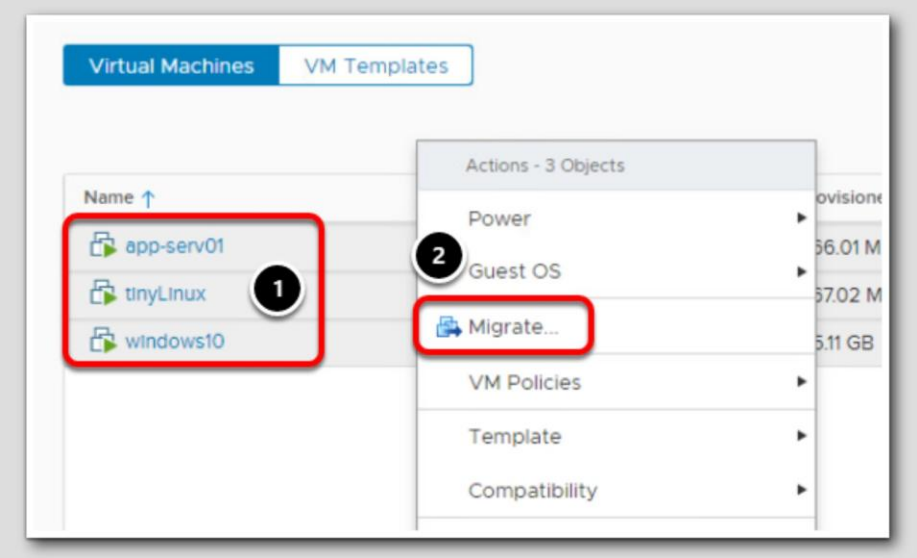
2.Right click and select Power/Power On

Do this for every powered off virtual machine, otherwise the next step will fail.



Migrate VMs

1. Select all the virtual machines (click the first one on the list, hold the shift key, click the last oneon the list).
2. Right click and select Migrate…



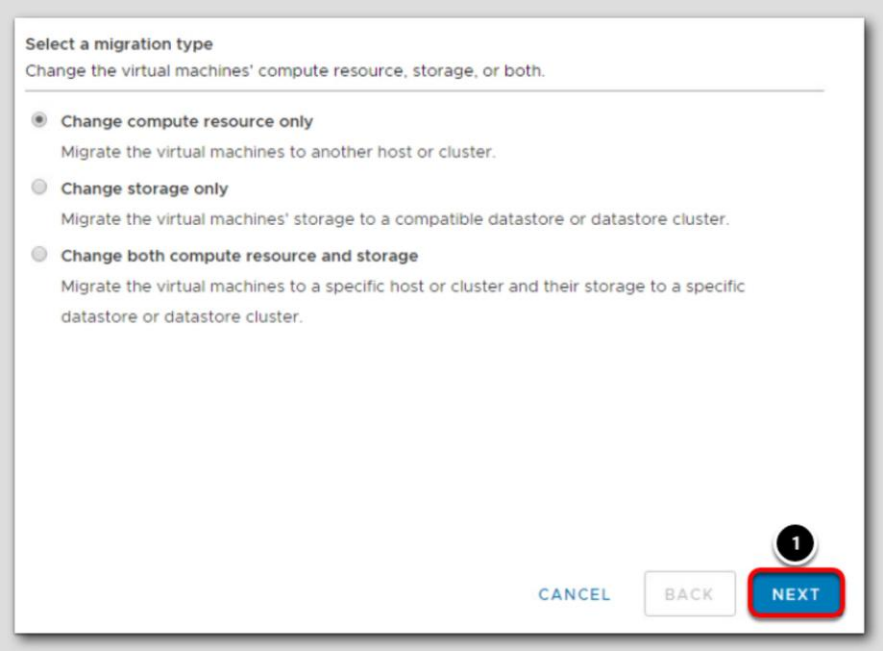
Migrate

Click Yes to start the migration process.

Migration Type

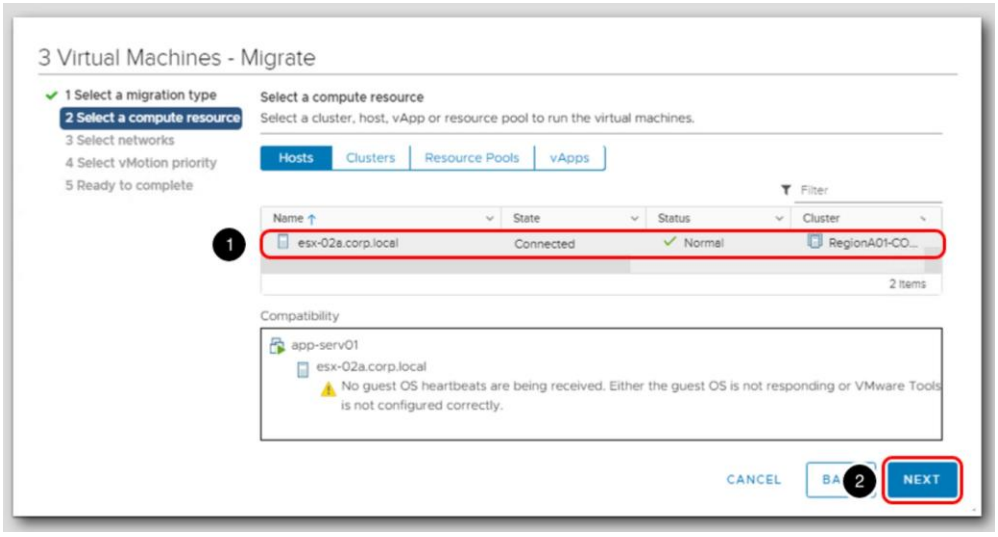
1. Leave the default setting and click Next.

In addition to changing what ESXi host the virtual machine will run on (using compute resources), the virtual machine can be moved to different datastores (storage) if needed, A virtual machine can also be moved to a different host and storage at the same time, More on migrating to different storage is covered in Module 3, in the Storage vMotion lesson



Compute Resource

1. Select esx-02a.corp.localesx-02a.corp.local
2. Click Next

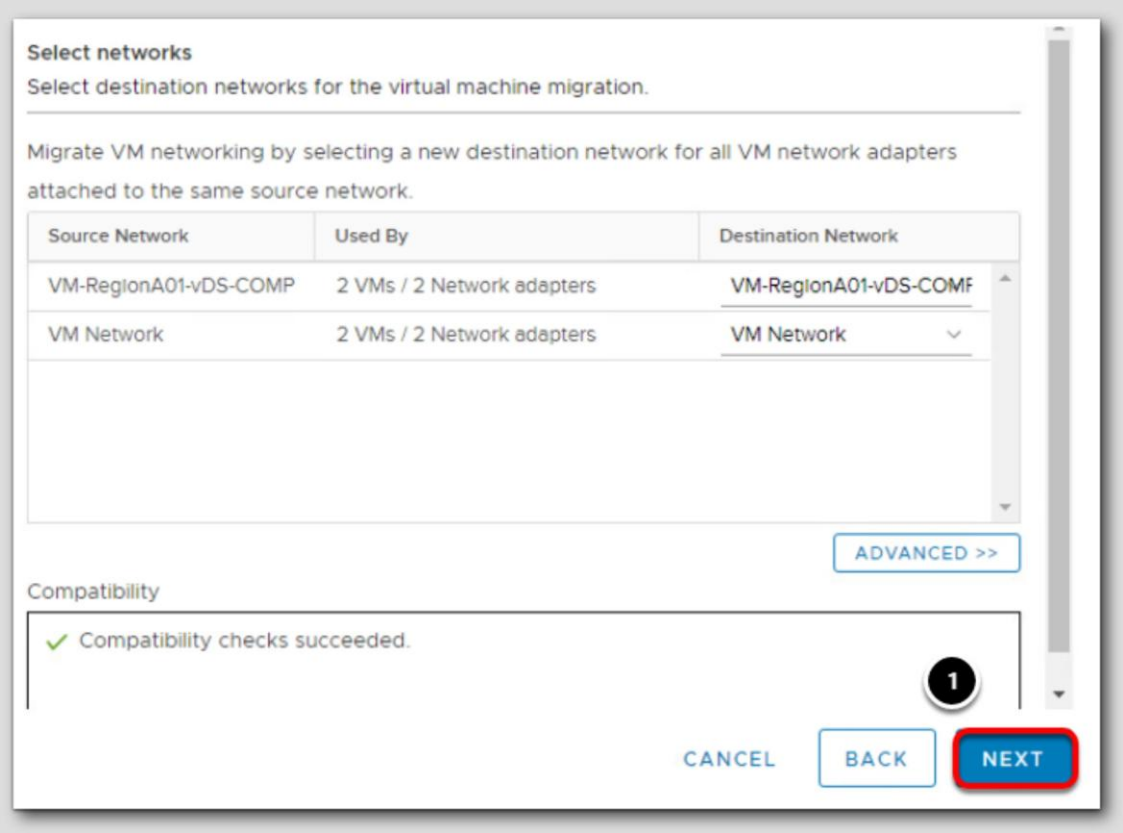


Since we want to move all the virtual machines to esx-02a.corp.local, we are selecting a specific host. We could also place it in a Cluster and let DRS decide the best host to move it to.

Networks

In most cases, the network adapter will not need to be changed.

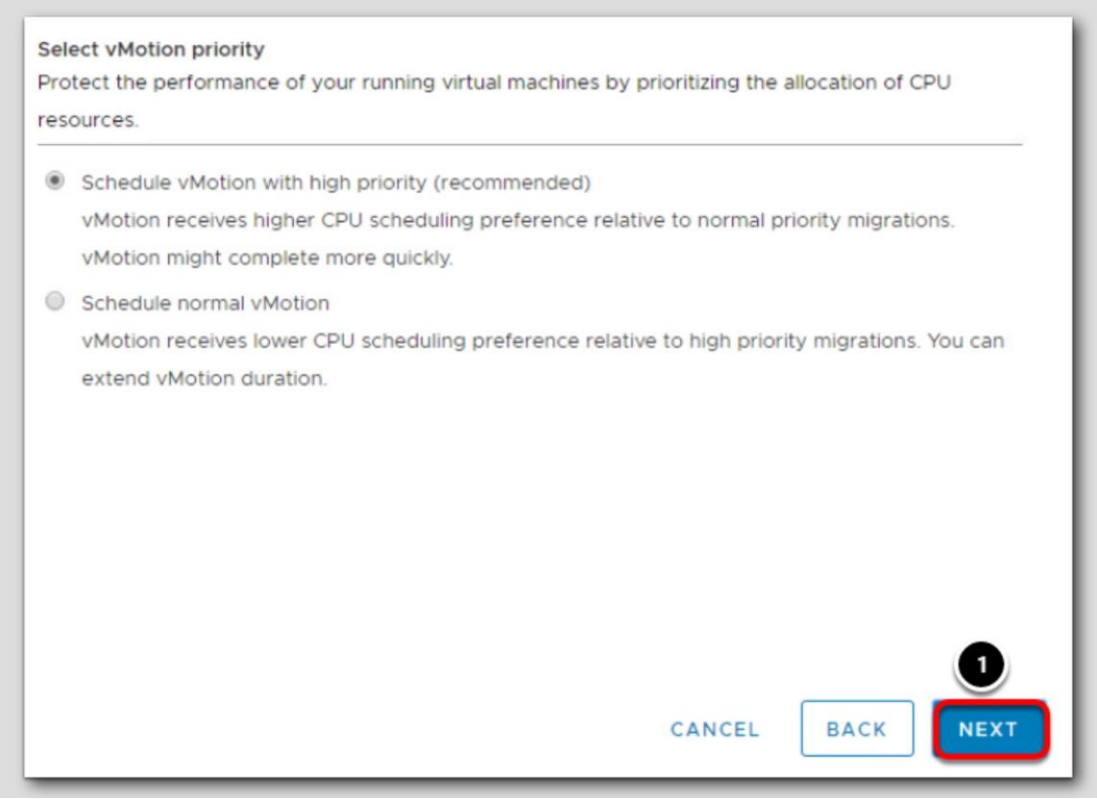
1. Click Next



vMotion Priority

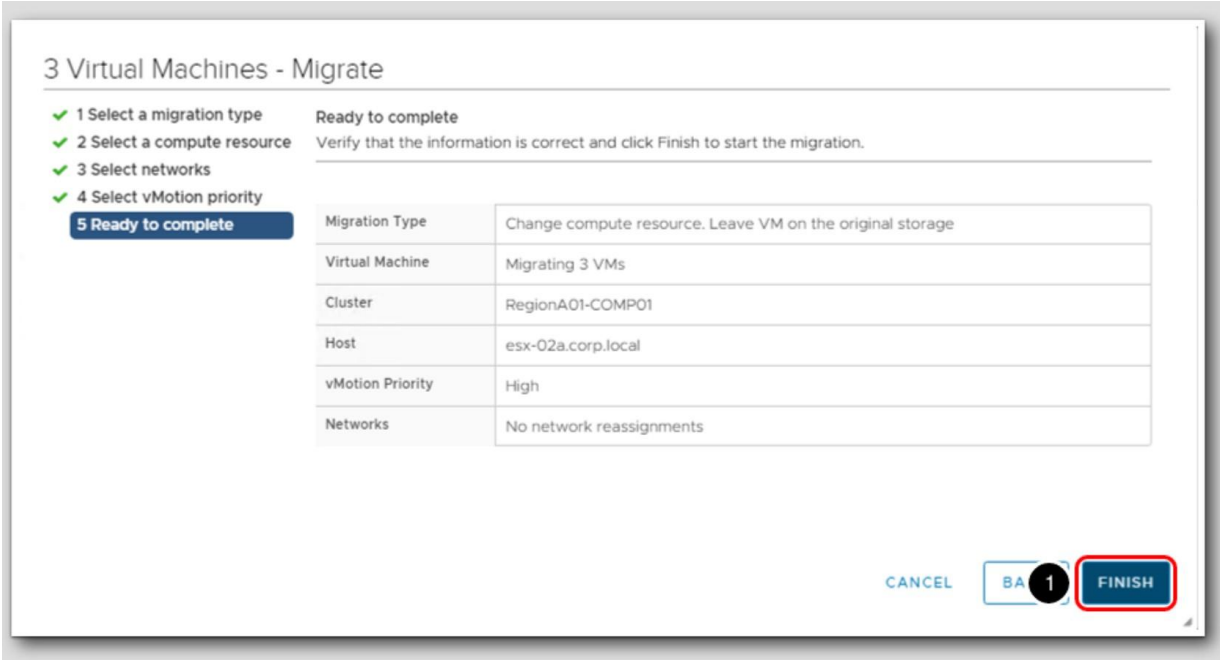
A priority can be set for the vMotion task. In most cases, the default option is OK.

1. Leave the default setting and click Next



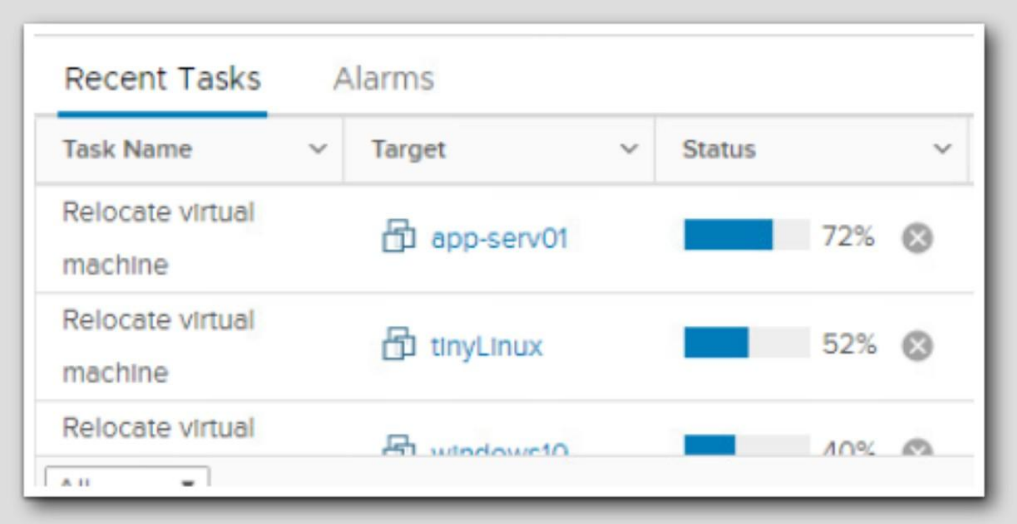
Ready to Complete

Review the settings and click Finish to migrate the virtual machines to esx-02a.corp.local.



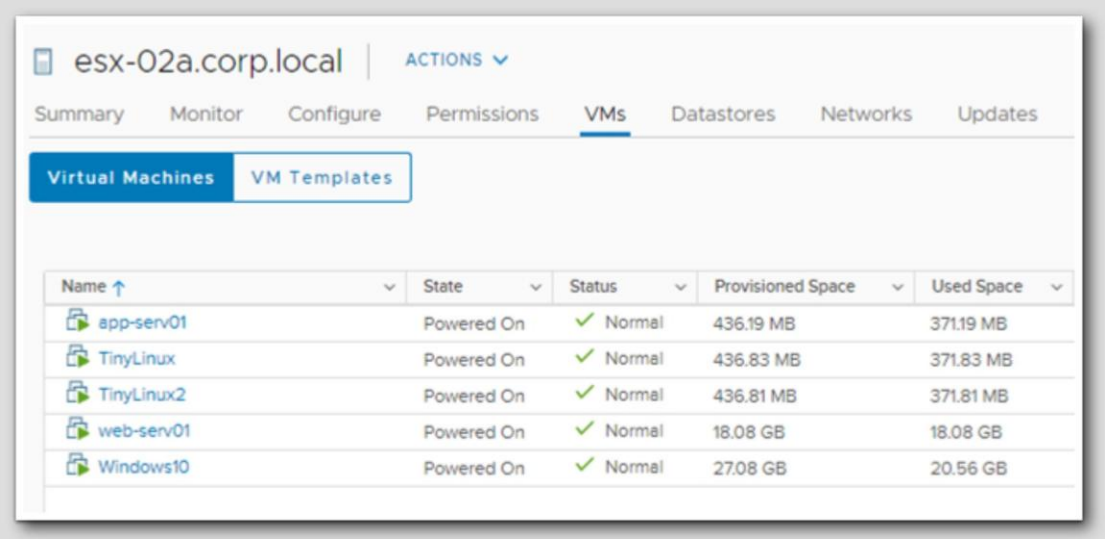
Monitor Progress

You can monitor progress using Recent Tasks.



Migration Complete

When the task has been completed successfully, you should see all of the virtual machines moved over to esx-02a.corp.local.

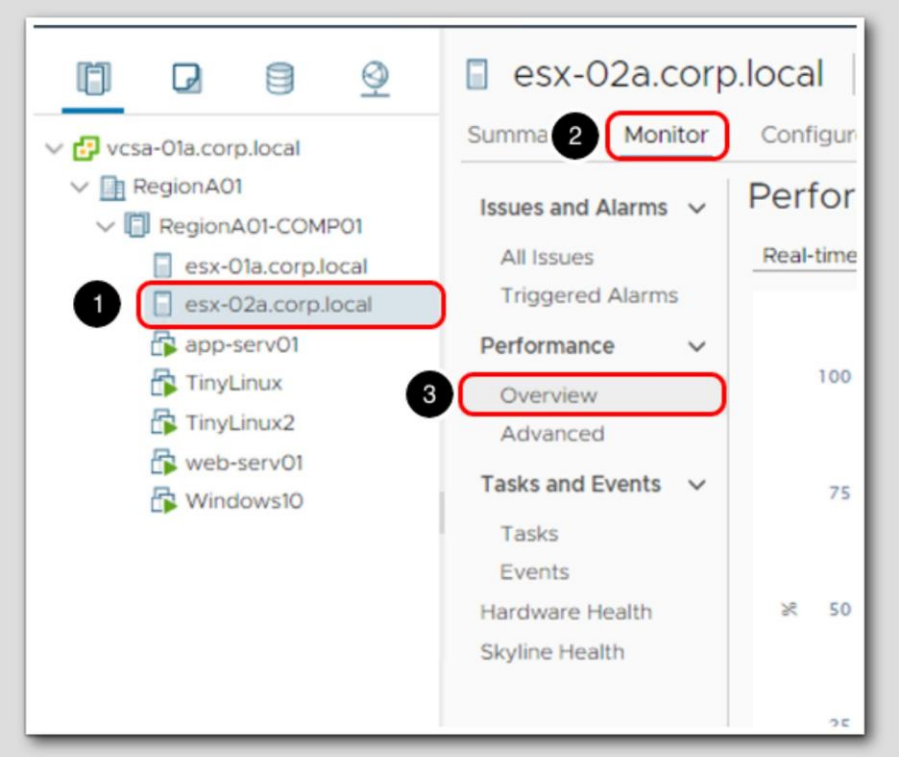


Practical 9: Use the system monitoring tools to reflect the CPU workload.

VMware provides several tools to help you monitor your virtual environment and to locate the source of potential issues and current problems. This lesson will walk through using the performance charts and graphs in the vSphere Client.

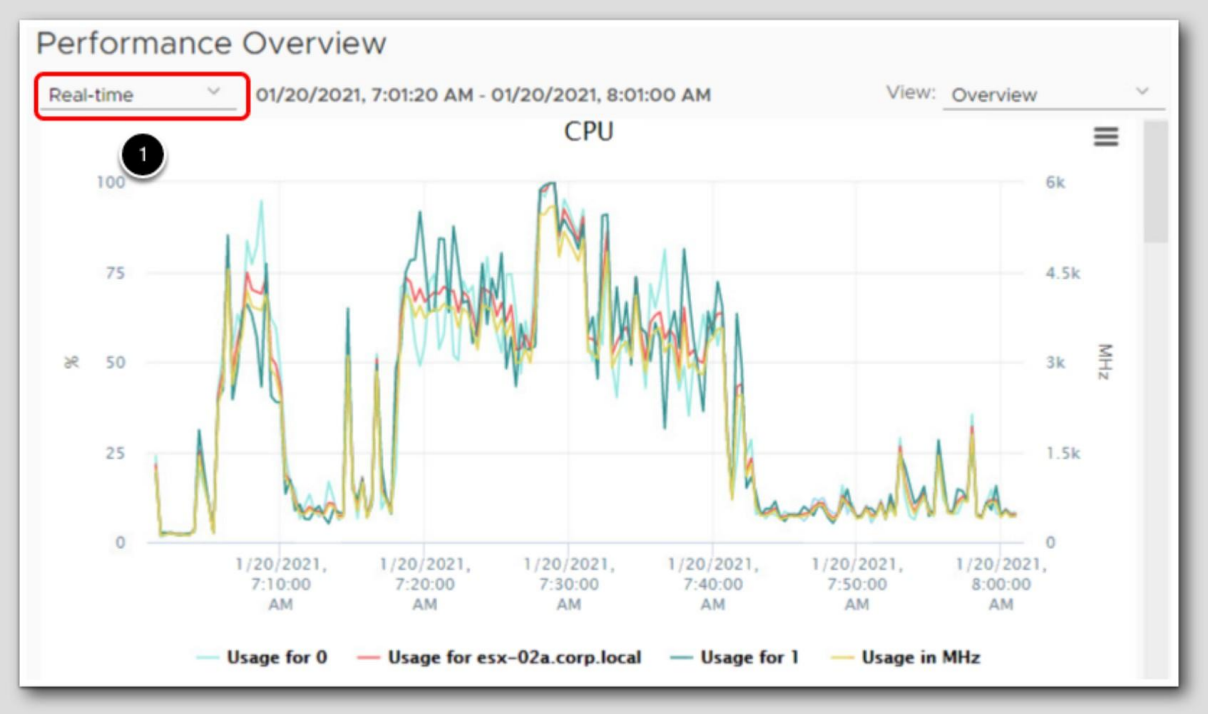
Select esx-02a

1. Select esx-02a.corp.local
2. Click the Monitor tab
3. Click Overview under the Performance section.



Host CPU Usage

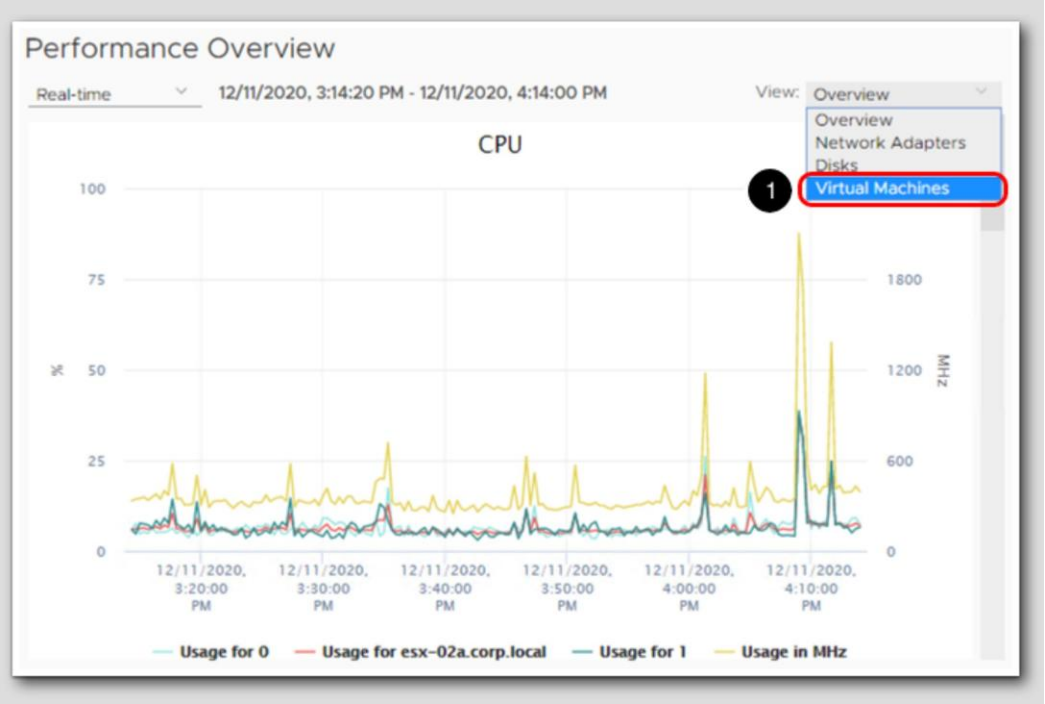
1. Ensure Real-timeReal-time has been selected from the Time Range drop-down menu



Here we can see in real time the CPU usage in percent for esx-02a.corp.local. By default, the chart will refresh every 20 seconds. The amount of data you see will depend on how long you have been taking the lab.

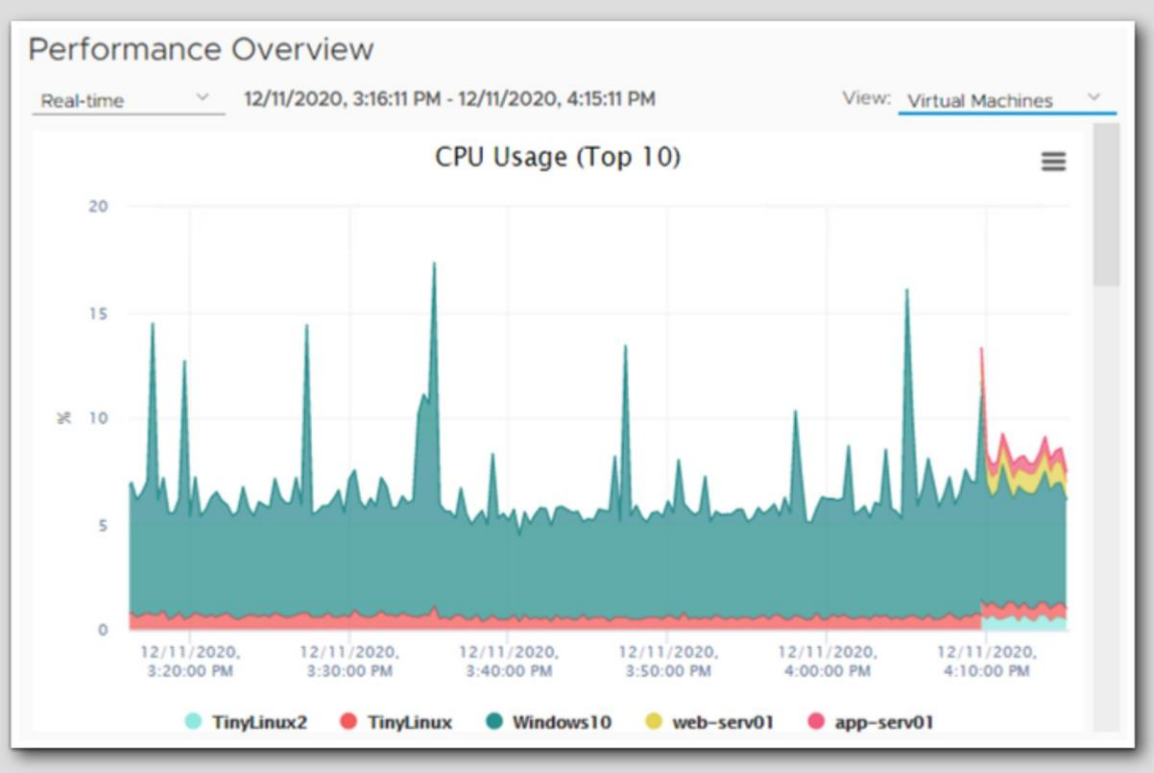
Virtual Machine CPU Usage

1. Now click the View drop-down box and select Virtual Machines.



Combined CPU Usage

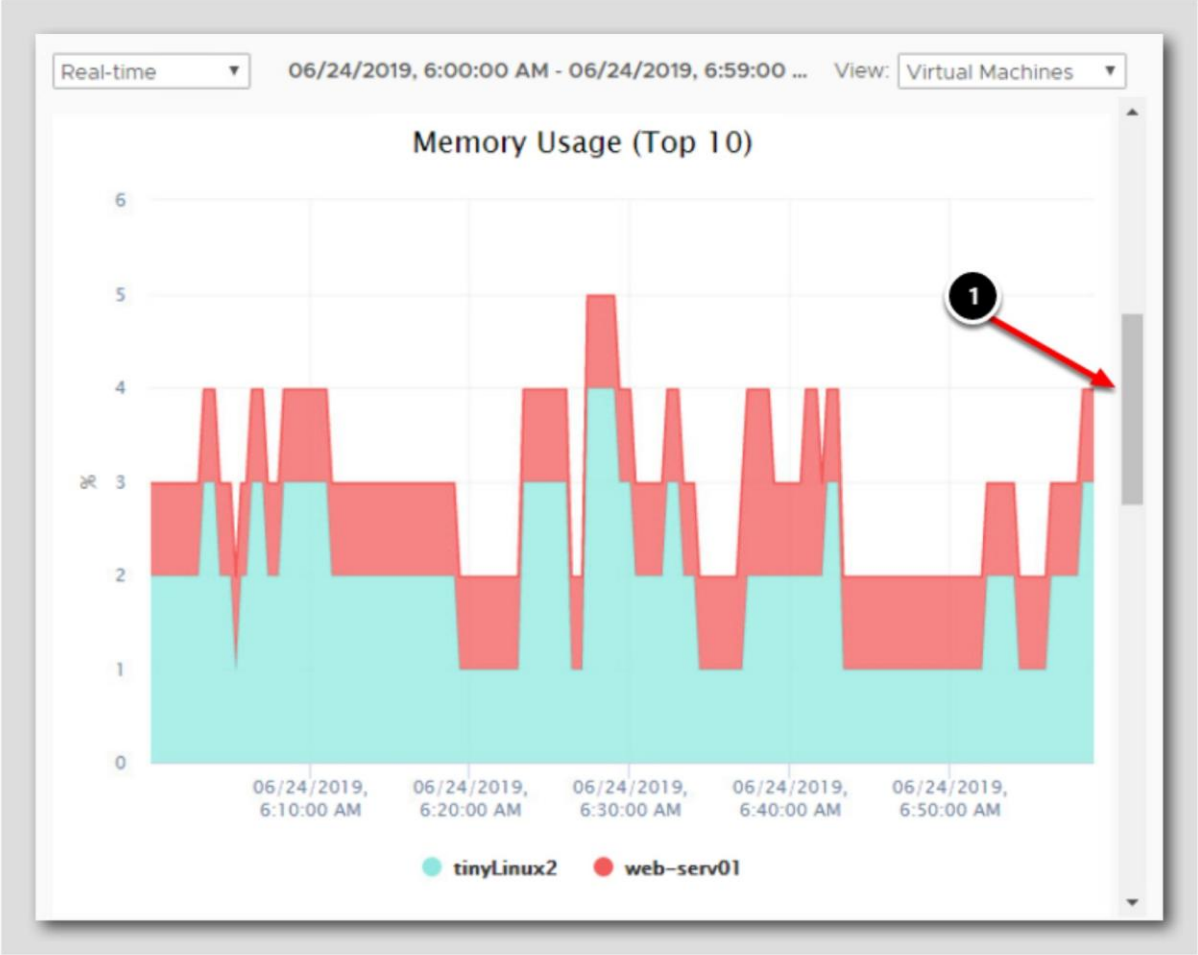
This chart shows the real-time CPU usage of each virtual machine. Each VM is represented by a different color in the graph and you can see at the bottom, which VM is represented by what color. Combined, they give you an idea of overall CPU usage on the host.



Other Available Graphs

There are other graphs available to show host and virtual machine memory usage, network (Mbps) and disk (KBps).

1. Use the scroll bars to access the additional charts.



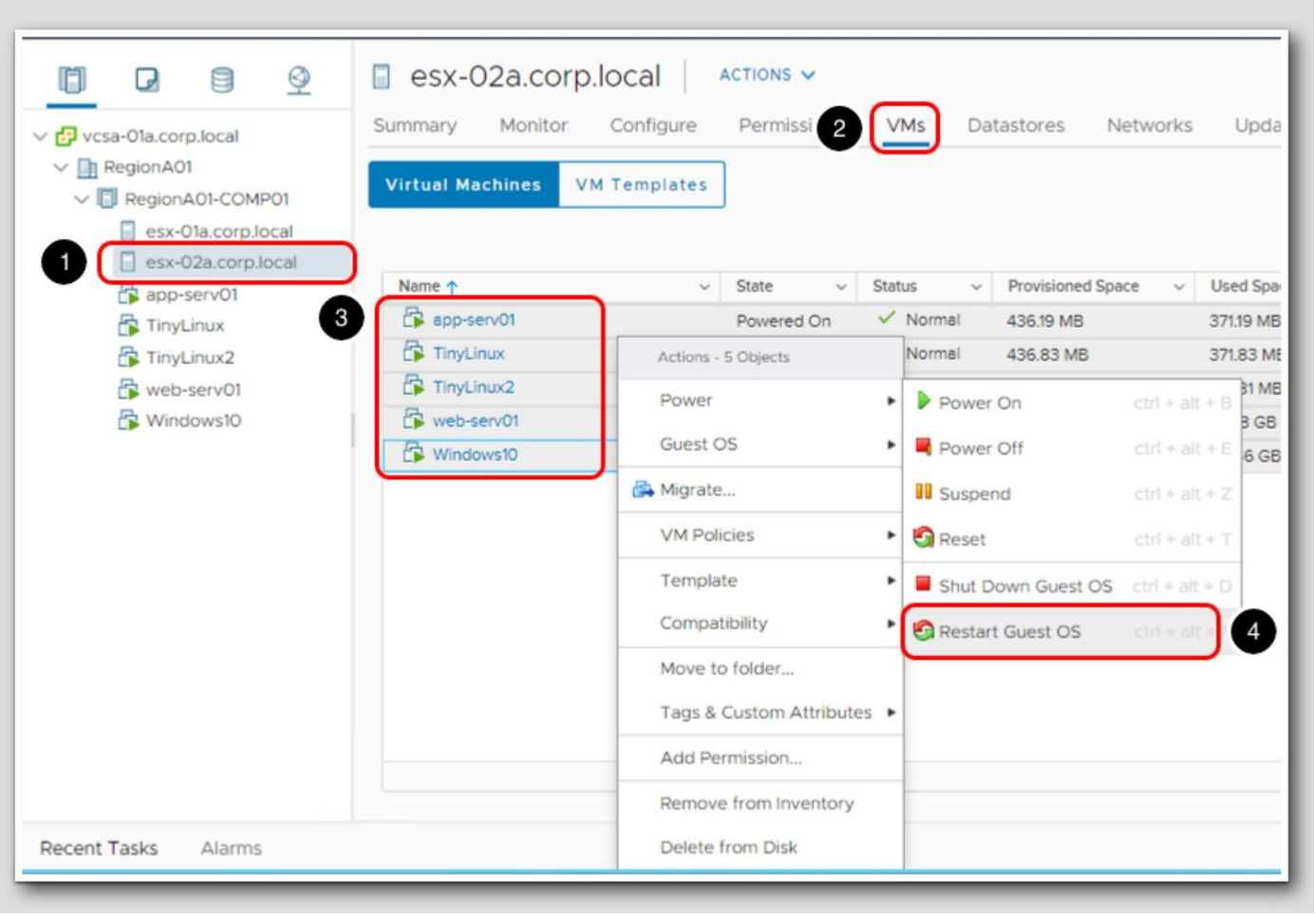
The graphs we have looked at so far will give you an overview of the four main components, CPU, memory, disk and storage. The advanced graphs will give you more detailed information on each of these.

Before we look at these charts, let's generate some CPU activity on esx-01a.corp.local by restarting all of the virtual machines it hosts

Select the VMs to be Restarted

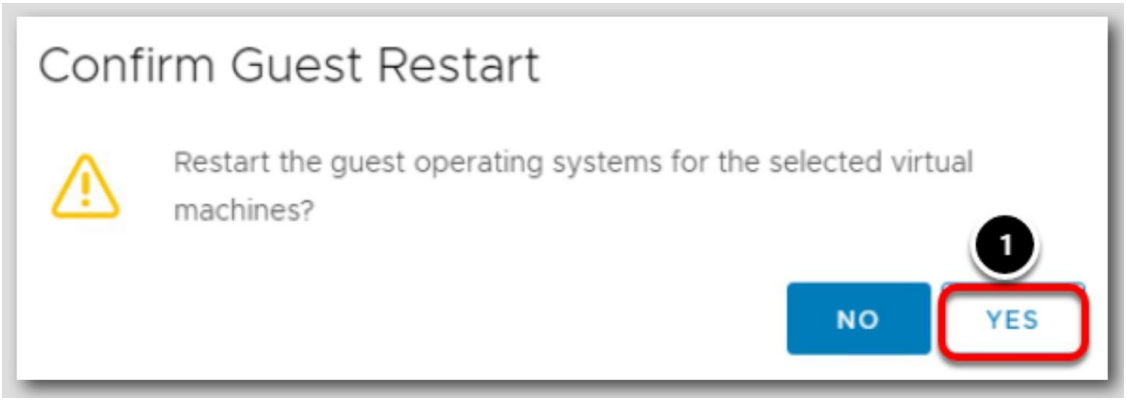
To generate some activity on esx-02a.corp.local, the virtual machines will be rebooted.

1. Select esx-02a.corp.local
2. Click on the VMs tab
3. Click on the first VM that is listed, hold down the Shift key and select the last VM on thelist.
4. Select Power and click the Restart Guest OS button.



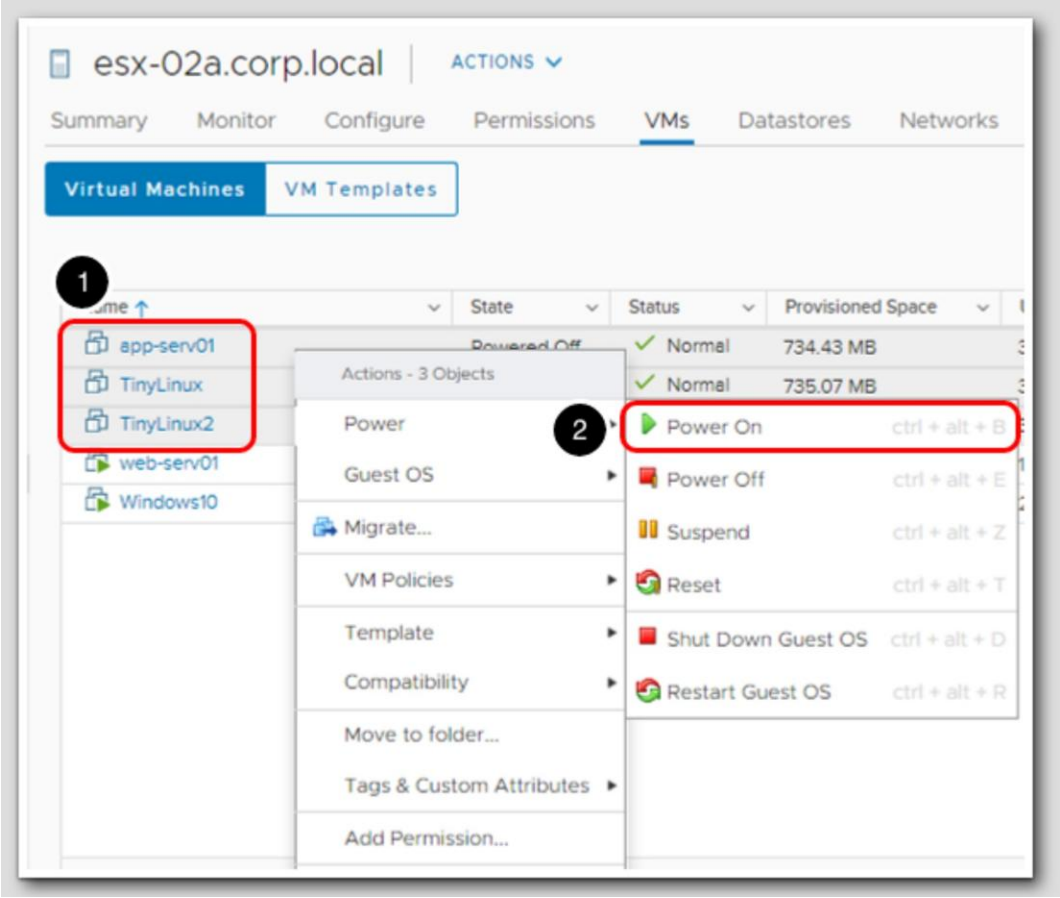
Confirm Restart

1. Click Yes to continue.



Note: You may also receive a warning that only X of X virtual machines will be restarted. This depend on what other modules and/or lessons have been completed in the lab previously. **Manually Start VMs**

1. If TinyLinux, TinyLinux2, or app-serv01 did not restart, but instead shut down.
2. Select all and power them on manually.



Monitor Performance

1. Click on the Monitor tab.
2. Click Advanced in the Performance section.

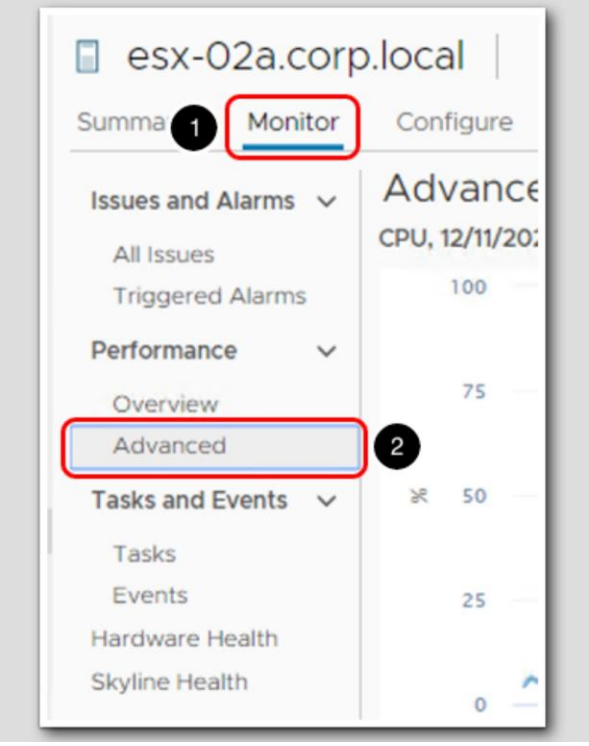
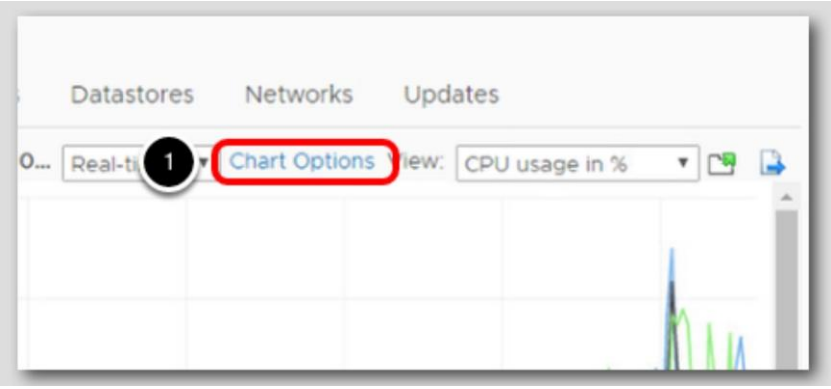


Chart Options

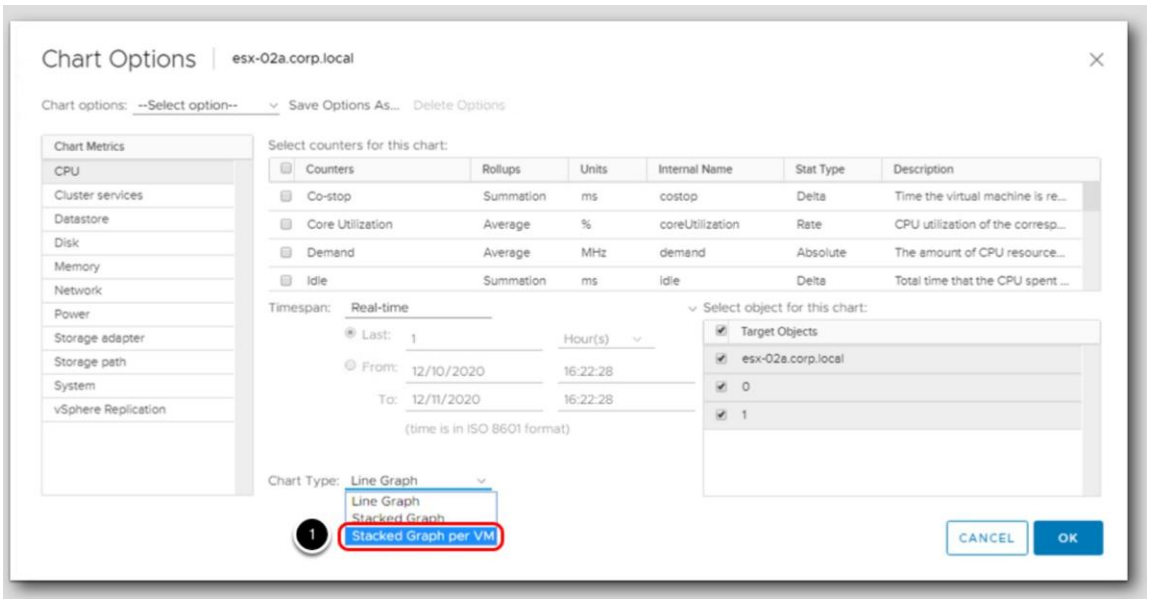
1. Click the Chart Options link.



This will bring up options to customize the chart.

Stacked Graph per VM

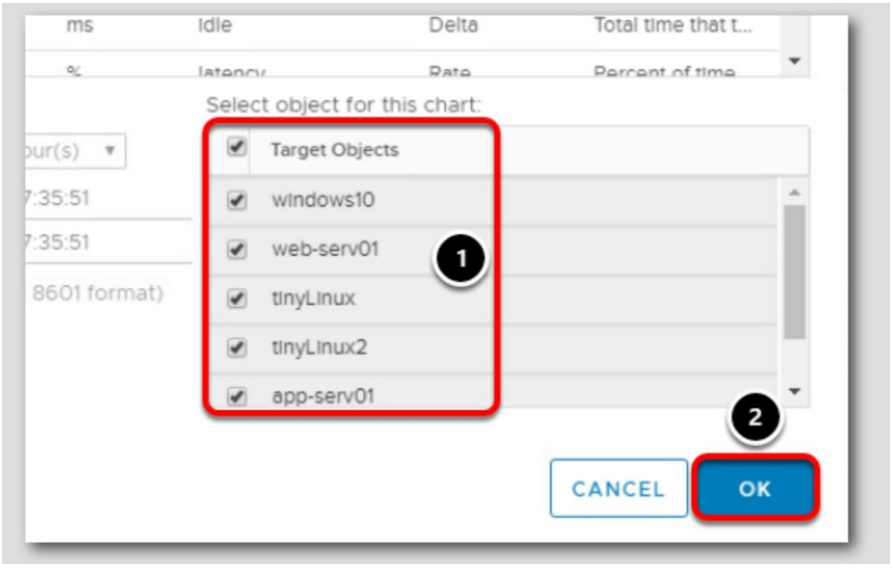
1. From the Chart Type drop-down menu, select Stacked Graph per VM.



Select Objects

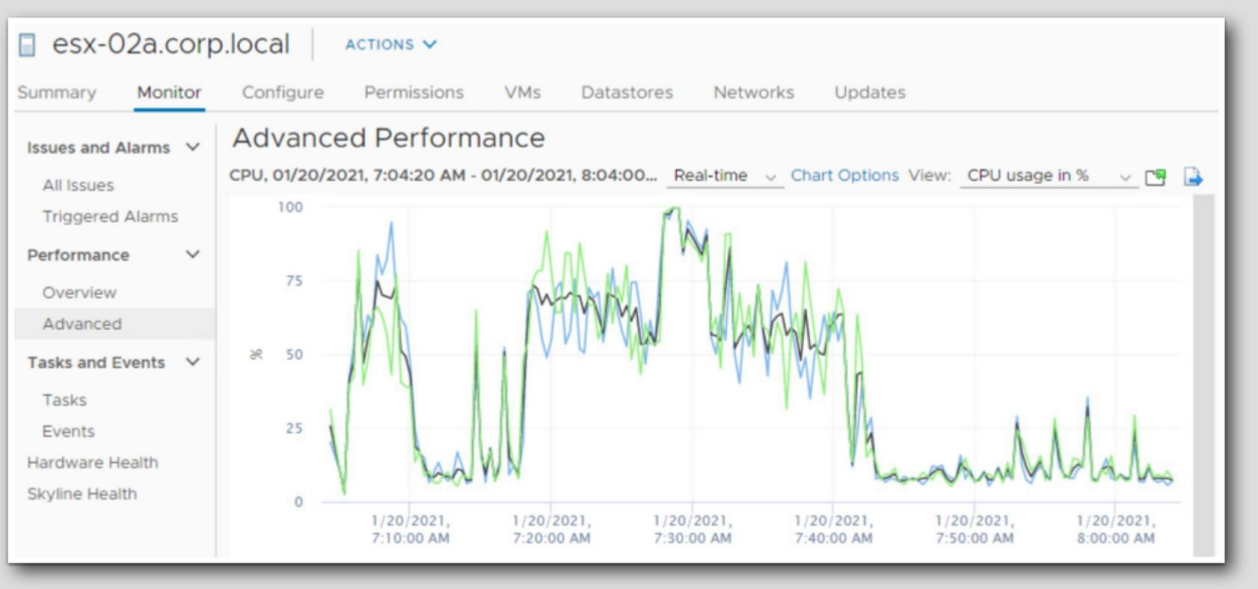
1. Under the Select objects for this chart box, verify all the virtual machines are selected.

2 .Click the OK button to see the newly customized chart.



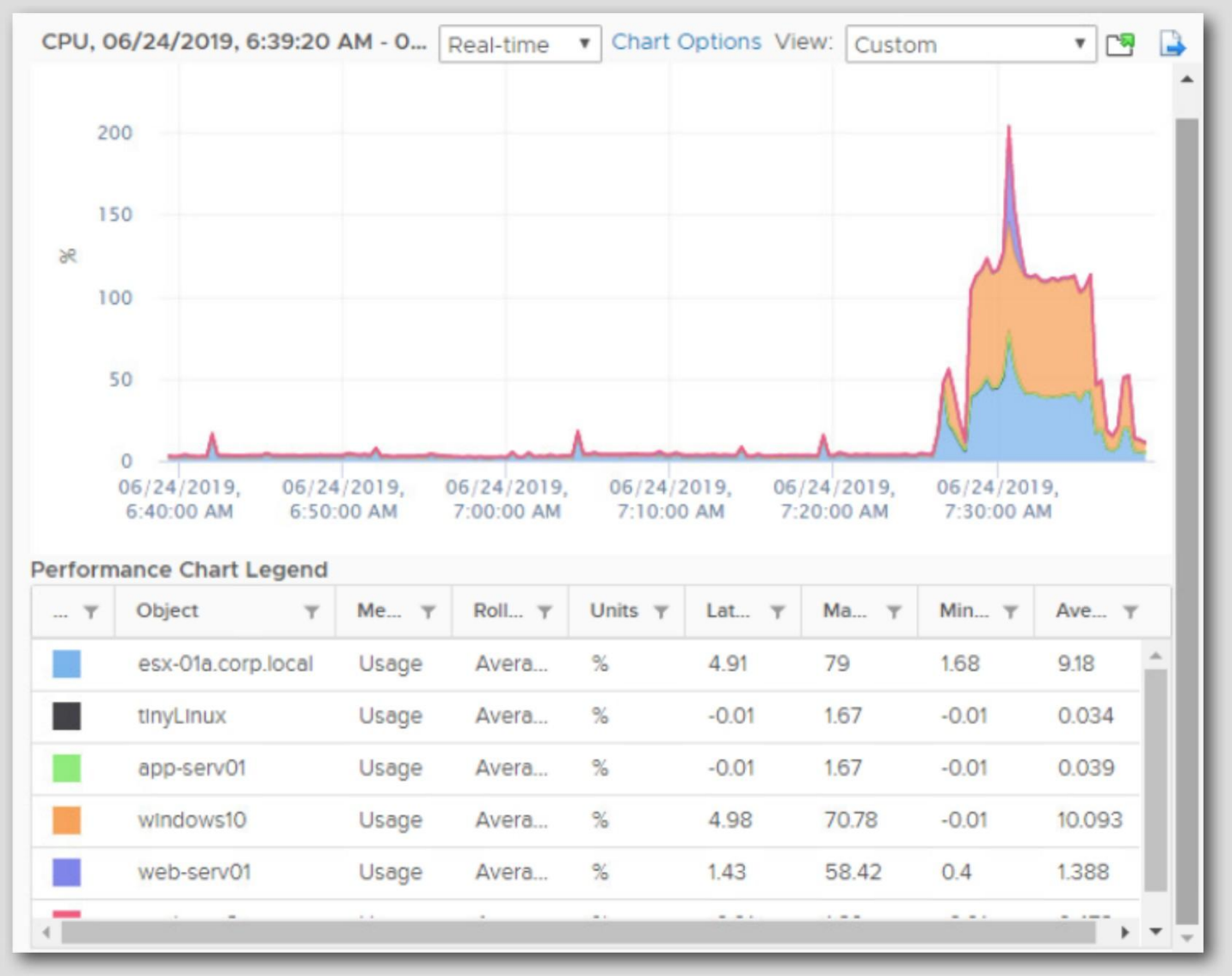
CPU Usage in Real-time

Here we can see the CPU usage of each virtual machine and esx-02a.corp.local.



Performance Chart Legend

Scroll down and you will see the Performance Chart Legend. You can click on any of the virtual machines or esx-01a.corp.local to highlight it on the chart.


CPU, 06/24/2019, 6:39:20 AM - 0...
Real-time
Chart Options
View:
Custom
200
150
%
100
50
0
06/24/2019, 6:40:00 AM
06/24/2019, 6:50:00 AM
06/24/2019, 7:00:00 AM
06/24/2019, 7:10:00 AM
06/24/2019, 7:20:00 AM
06/24/2019, 7:30:00 AM
Performance Chart Legend
... Object Me... Roll... Units Lat... Ma... Min... Ave...
esx-01a.corp.local Usage Avera... % 4.91 79 1.68 9.18
tinyLinux Usage Avera... % -0.01 1.67 -0.01 0.034
app-serv01 Usage Avera... % -0.01 1.67 -0.01 0.039
windows10 Usage Avera... % 4.98 70.78 -0.01 10.093
web-serv01 Usage Avera... % 1.43 58.42 0.4 1.388

Practical 10: Use the vCenter Server Appliance alarm feature.

vSphere includes a user-configurable events and alarms subsystem. This subsystem tracks events happening throughout vSphere and stores the data in log files and the vCenter Server database. This subsystem also enables you to specify the conditions under which alarms are triggered.

Alarms can change state from mild warnings to more serious alerts as system conditions change and can trigger automated alarm actions. This functionality is useful when you want to be informed, or take immediate action, when certain events or conditions occur for a specific inventory object, or group of objects.

Events are records of user actions or system actions that occur on objects in vCenter Server or on a host. Actions that might be reordered as events include, but are not limited to, the following examples:

- A license key expires
- A virtual machine is powered on
- A user logs in to a virtual machine
- A host connection is lost

Event data includes details about the event such as who generated it, when it occurred, and what type of event.

Alarms are notifications that are activated in response to an event, a set of conditions, or the state of an inventory object. An alarm definition consists of the following elements ● Name and description - Provides an identifying label and description.

- Alarm type - Defines the type of object that will be monitored.
- Triggers - Defines the event, condition, or state that will trigger the alarm and defines the notification severity.
- Tolerance thresholds (Reporting) - Provides additional restrictions on condition and state triggers thresholds that must be exceeded before the alarm is triggered.
- Actions - Defines operations that occur in response to triggered alarms. VMware provides sets of predefined actions that are specific to inventory object types.

Alarms have the following severity levels:

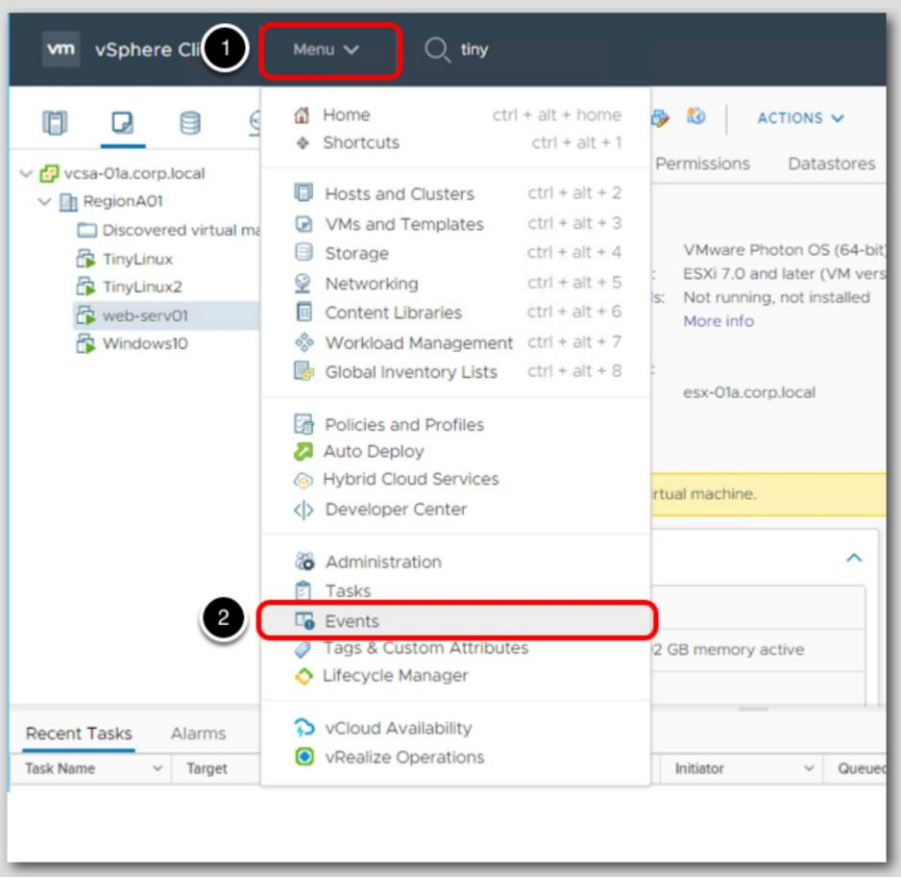
- Normal – green
- Warning – yellow
- Alert – red

Alarm definitions are associated with the object selected in the inventory. An alarm monitors the type of inventory objects specified in its definition

For example, you might want to monitor the CPU usage of all virtual machines in a specific host cluster. You can select the cluster in the inventory and add a virtual machine alarm to it. When enabled, that alarm will monitor all virtual machines running in the cluster and will trigger when any one of them meets the criteria defined in the alarm. If you want to monitor a specific virtual machine in the cluster, but not others, you would select that virtual machine in the inventory and add an alarm to it. One easy way to apply the same alarms to a group of objects is to place those objects in a folder and define the alarm on the folder. In this lab, you will learn how to create an alarm and review the events that have occurred.

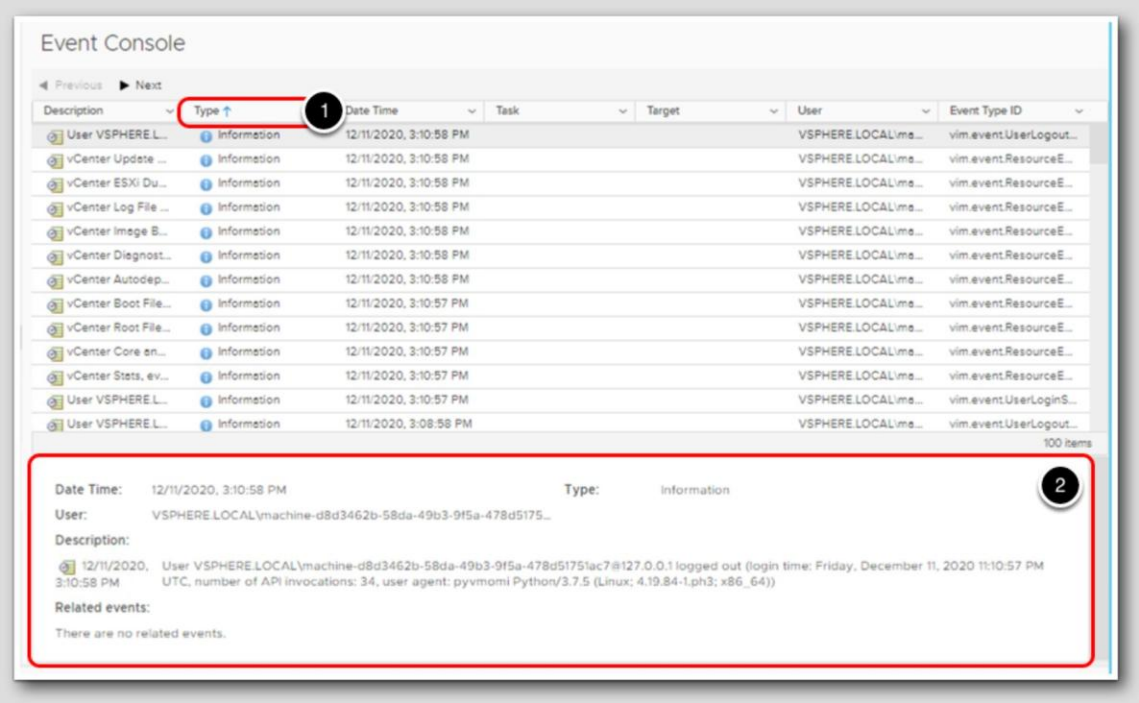
Review default alerts

1. Click Menu
2. Click on Events menu item



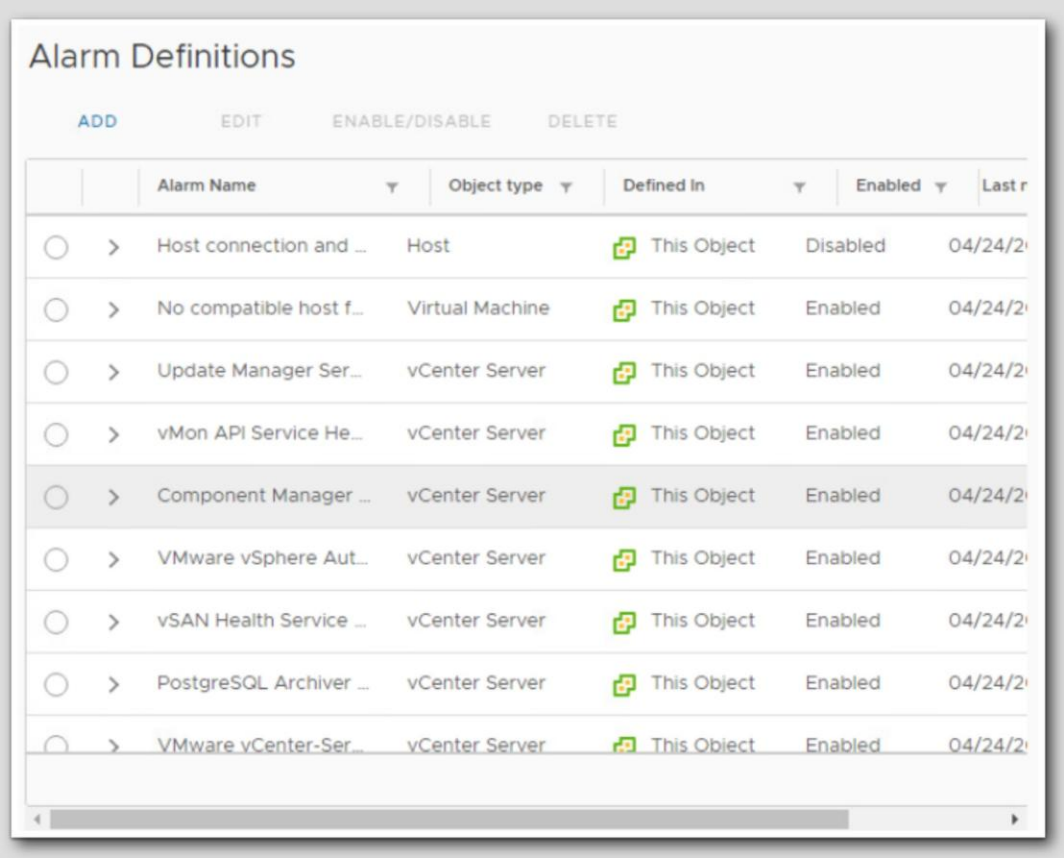
Event Console

1. Click on the TypeType column to sort by level of severity.
2. Select an event to review the details of the event.



Alarm Definitions

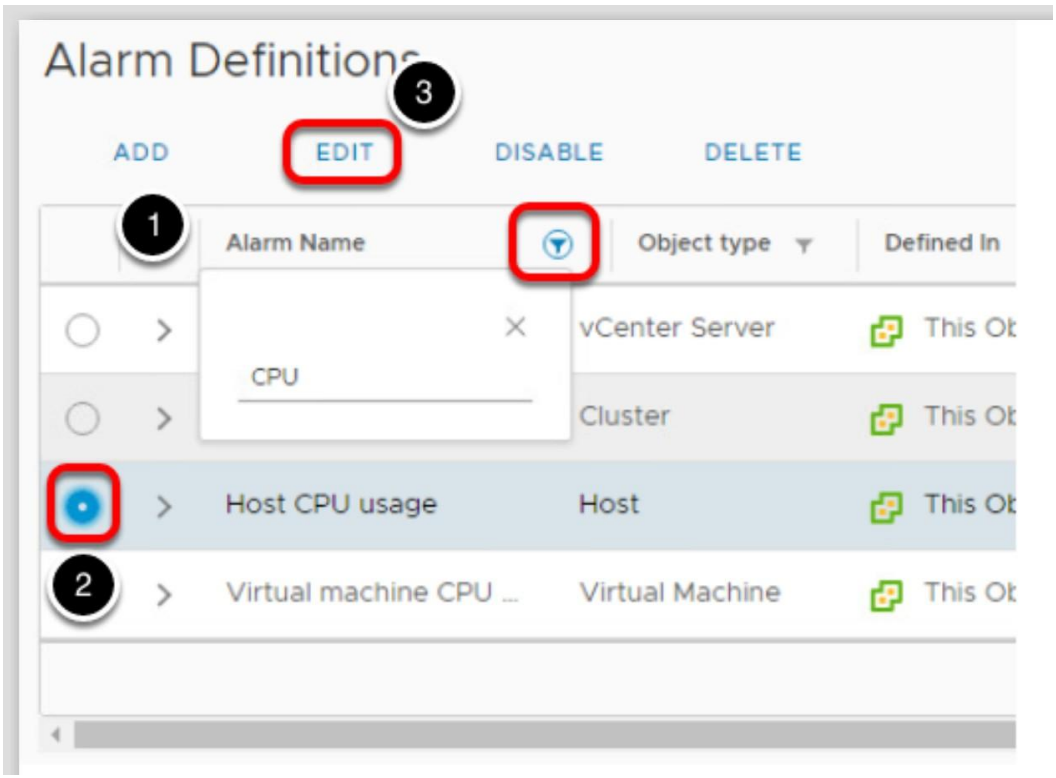
Alarms can be defined at different levels. In the case of the highlighted alarm, you can see it is defined at the top level (vCenter Server). Alarms that are defined at the top level are then inherited by the objects below.



Defining an Alarm

1. Click on the Alarm Name filter field and type cpucpu in the search field. 2. Select the Host CPU usage alarm

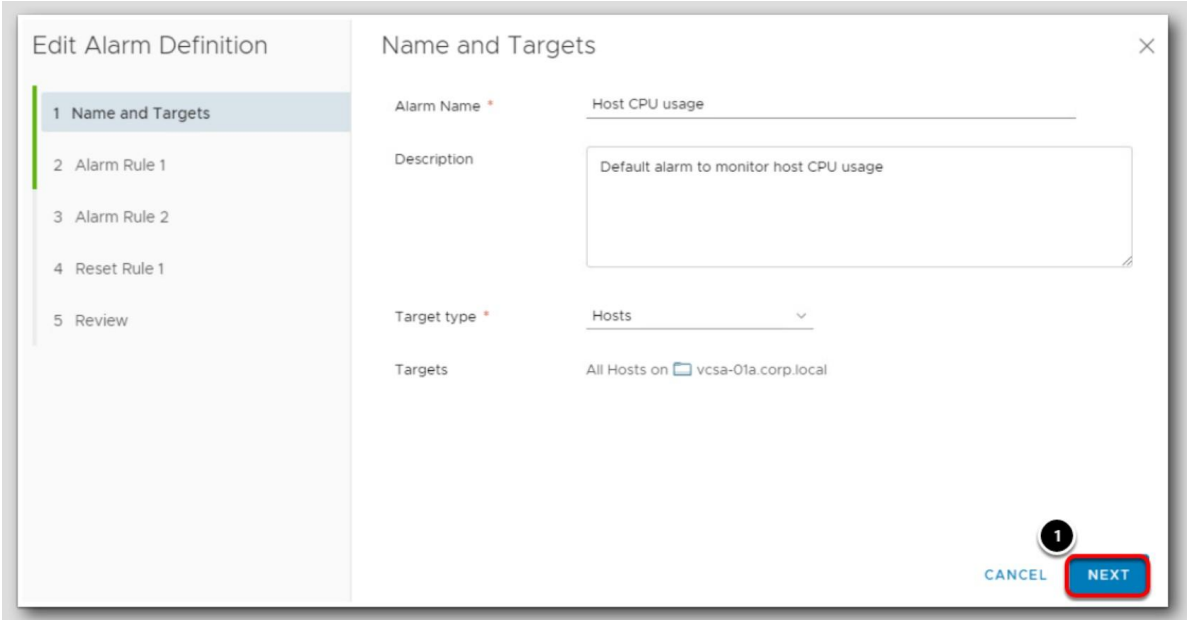
3. Click the Edit button.



Name and Targets

The Name and Targets screen defines the name of the alarm (Host CPU usage), what object it applies to (Hosts) and where the objects are located.

1. Click Next.



Alarm Rule 1

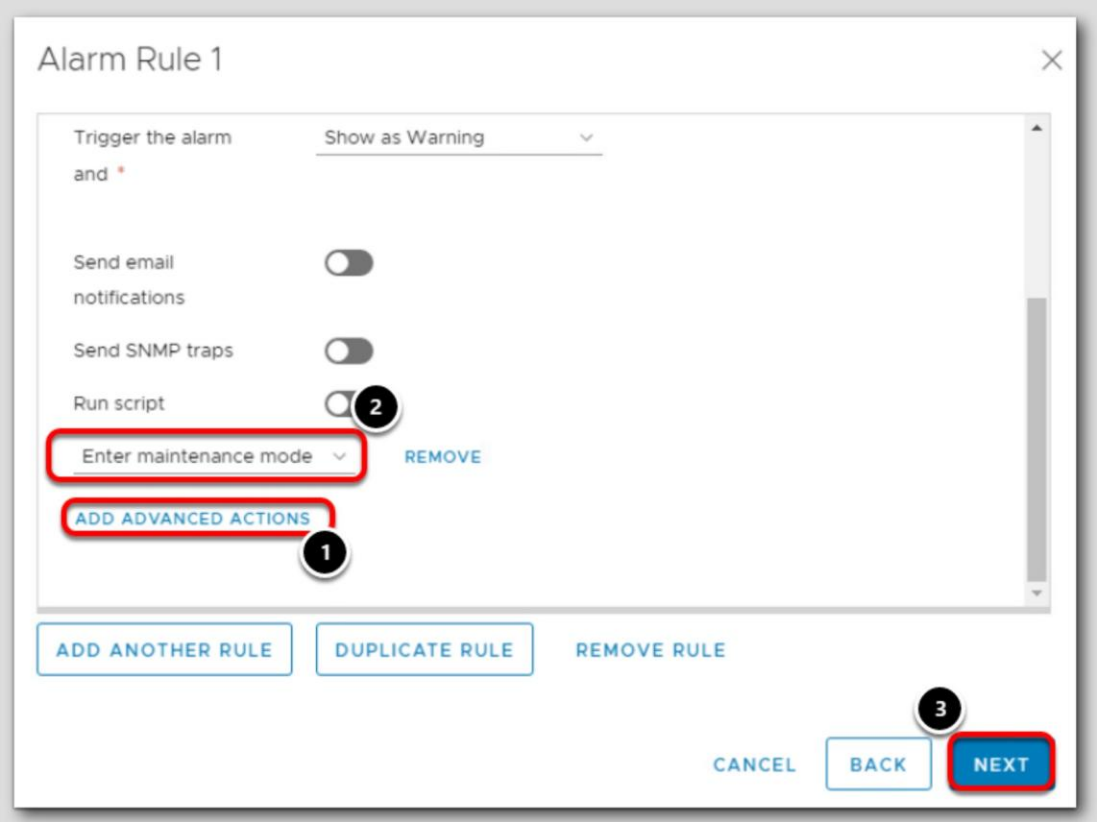
1. Change the percentage of 75% to 80%

2.Use the scroll bar to scroll to the bottom.

Notice this will trigger a Warning alarm.

Add Advanced Action

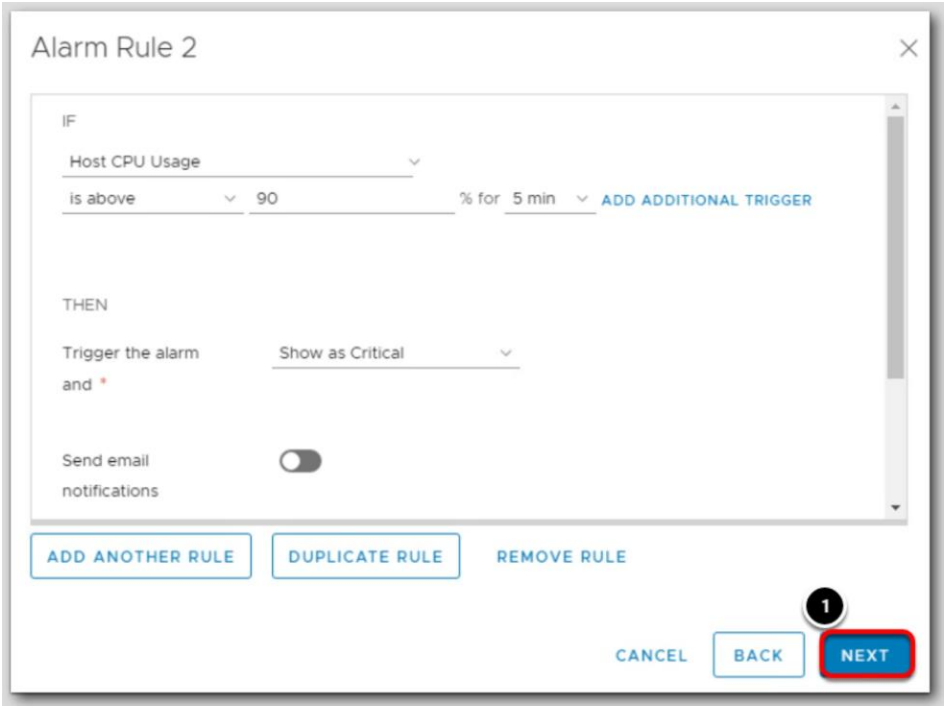
1. Click on Add Advanced Action.
2. From the drop-down menu (Select an advanced action), select Enter maintenance mode.
3. Click Next



When a Host's CPU runs at or above 80% for more than 5 minutes, a Warning alarm will be triggered, and the Host will be put in Maintenance mode.

Alarm Rule 2

1. Click Next.

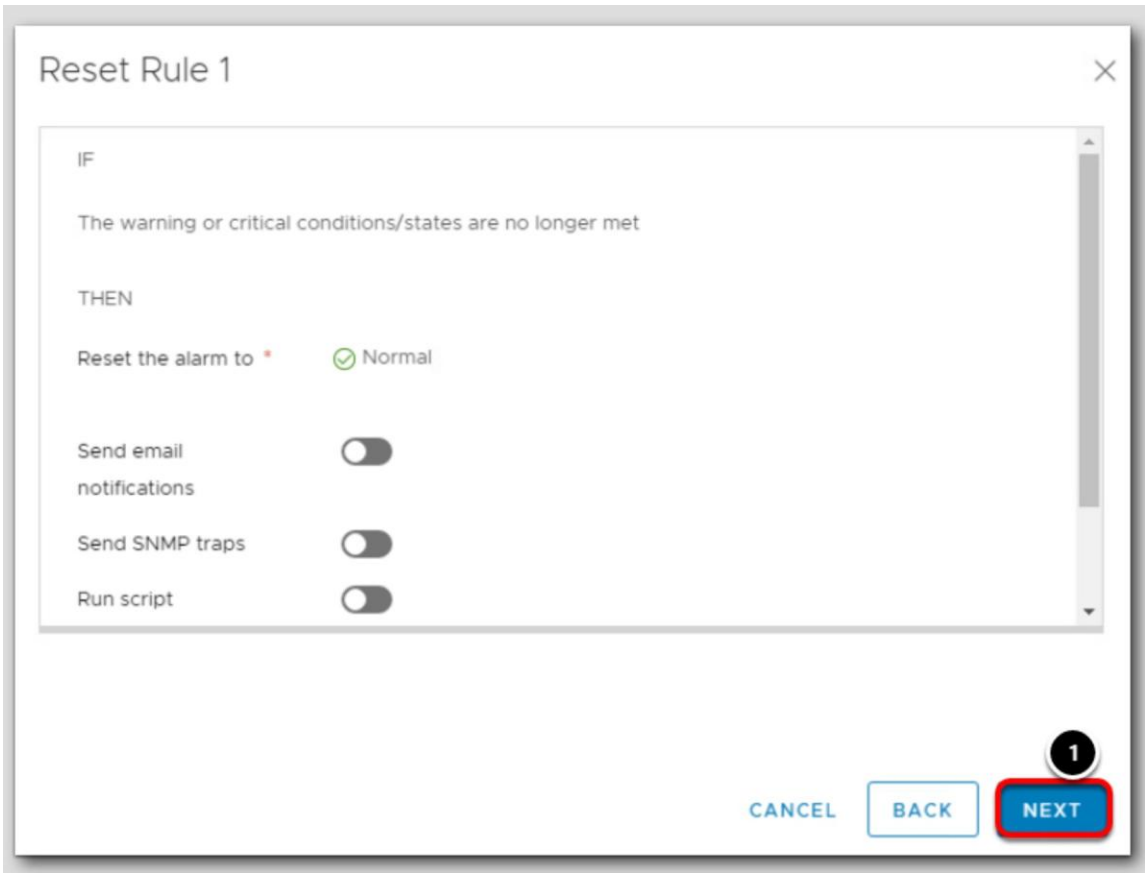


On this screen we can set additional actions based on when a Host's CPU is about 90% for 5 minutes. In this case, it would trigger a Critical alarm. Additional actions could be taken when a Host is in this state.

Reset Rule 1

If the conditions that originally triggered the alarm are no longer present, additional actions can take place. As an example, once a Host's CPU is no longer at 80% for more than 5 minutes, an email notification could be sent.

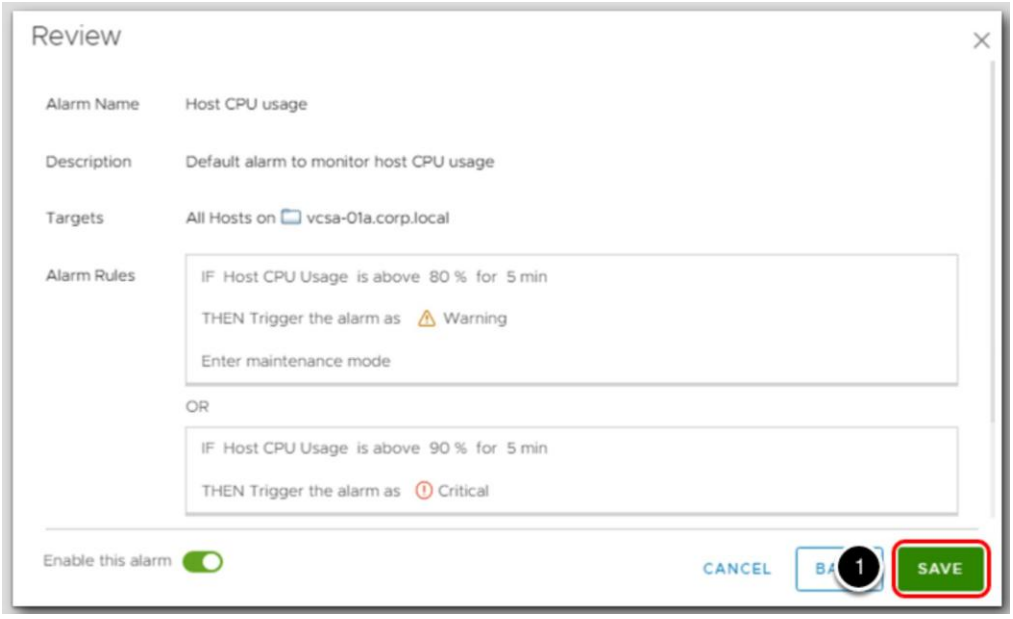
1. Click Next.



Review

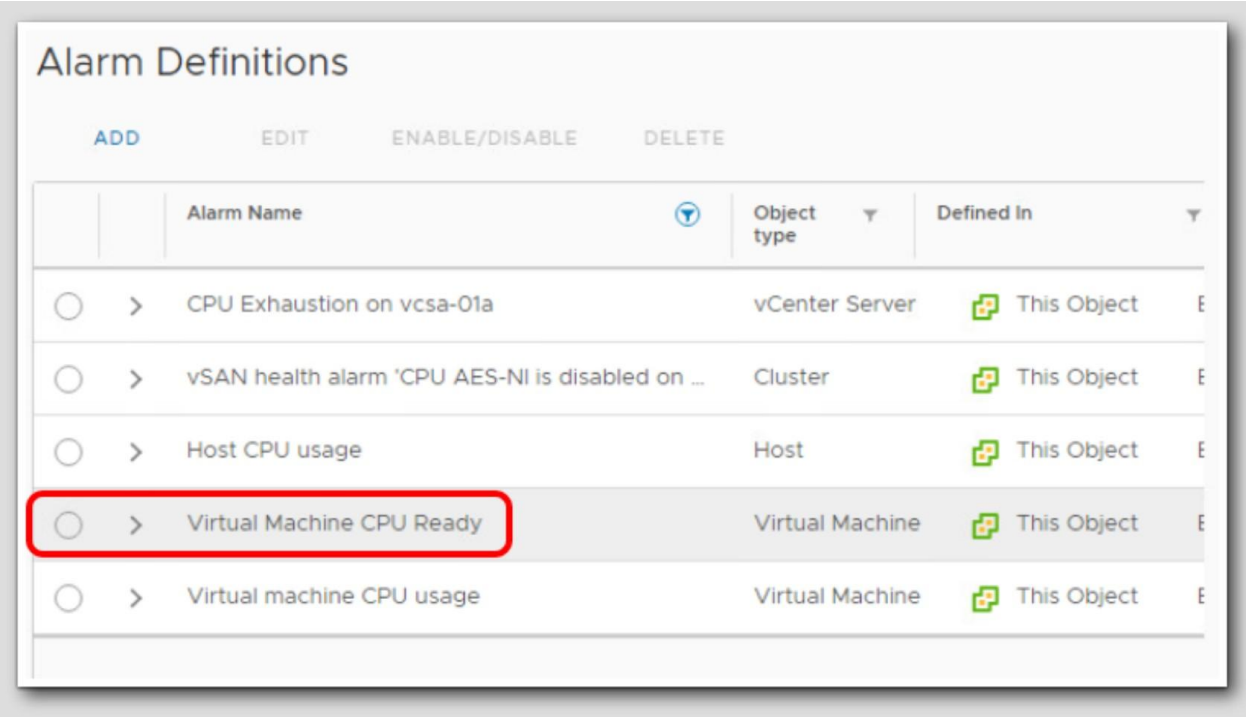
The Review screen shows what was configured.

1. Click Save to keep the changes made to the Alarm



New Alarm Created

If the Alarm Name field is still filtering by "cpu", the newly created alarm is displayed. If not, simply click on the Alarm Name field and type cpu ready to see it



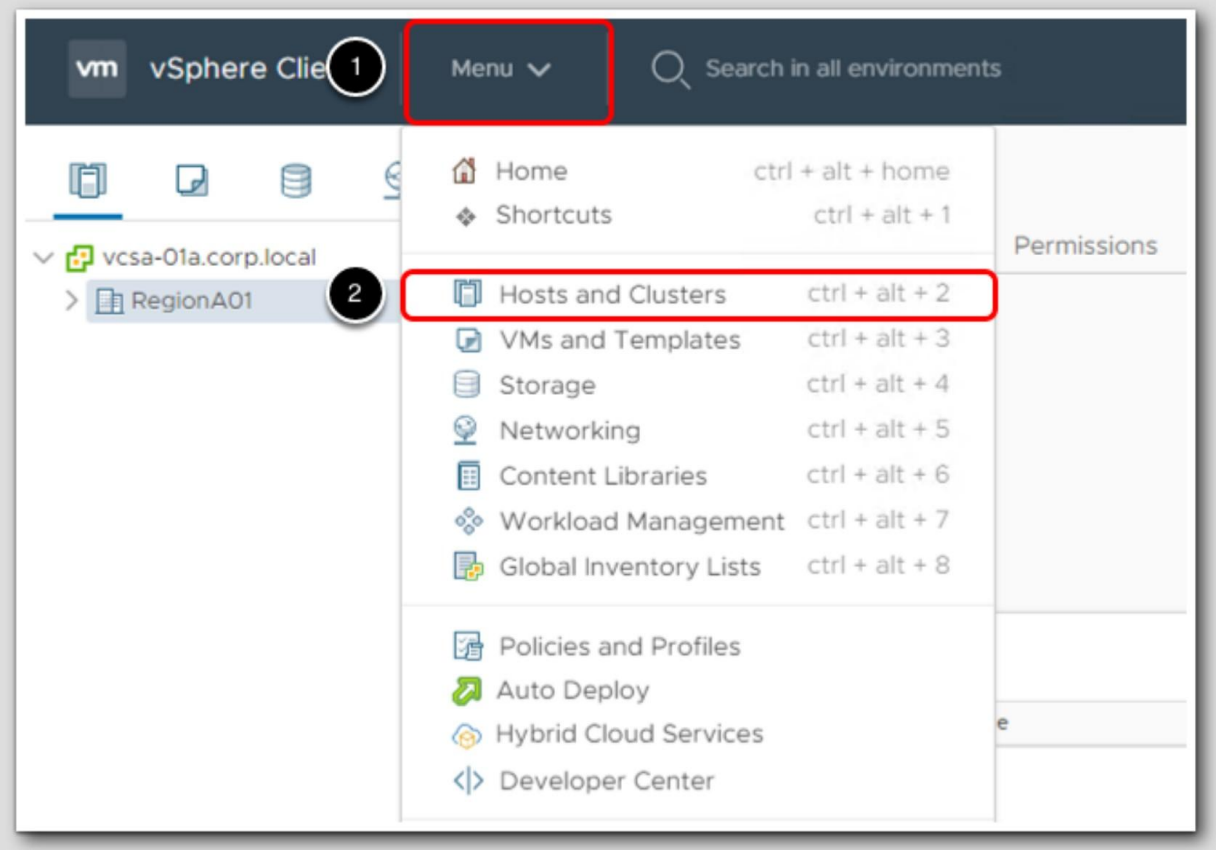
Practical 11: Use vSphere HA functionality.

vSphere Availability provides high availability for virtual machines by pooling the virtual machines and the hosts they reside on into a cluster. Hosts in the cluster are monitored and in the event of a failure, the virtual machines on a failed host are restarted on alternate hosts.

When you create a vSphere Availability cluster, a single host is automatically elected as the master host. The master host communicates with vCenter Server and monitors the state of all protected virtual machines and of the slave hosts. Different types of host failures are possible, and the master host must detect and appropriately deal with the failure. The master host must distinguish between a failed host and one that is in a network partition or that has become network isolated. The master host uses network and datastore heartbeating to determine the type of failure. Also note that vSphere Availability is a host function which means there is not a dependency on vCenter in order to effectively fail over VMs to other hosts in the cluster.

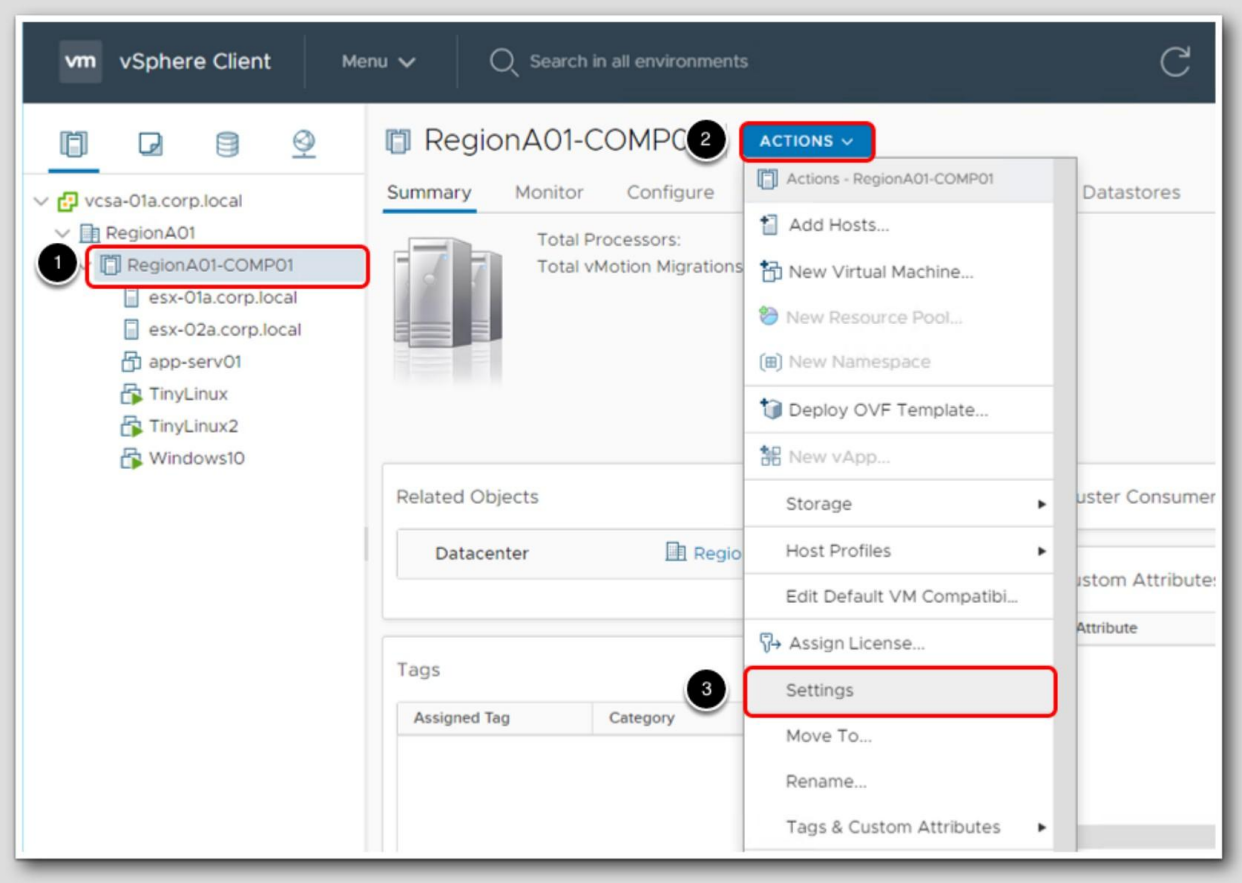
Enable and Configure vSphere Availability

1. First, click on Menu.
2. Select Hosts and Clusters.



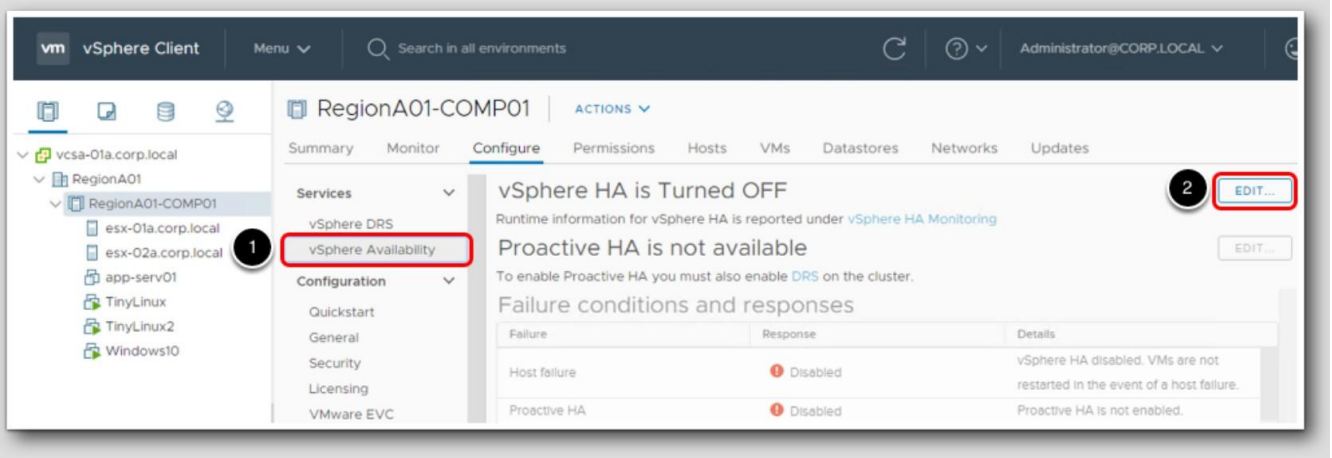
Settings for vSphere Availability

1. Click RegionA01 Cluster.
2. Click Actions to bring up the drop down-menu.
3. Click Settings.



Cluster Settings

1. Click vSphere Availability under Services to bring up the settings for high availability. Note thatyou may need to scroll to the top of the list.
2. Click the Edit button next to vSphere HA is Turned OFF.

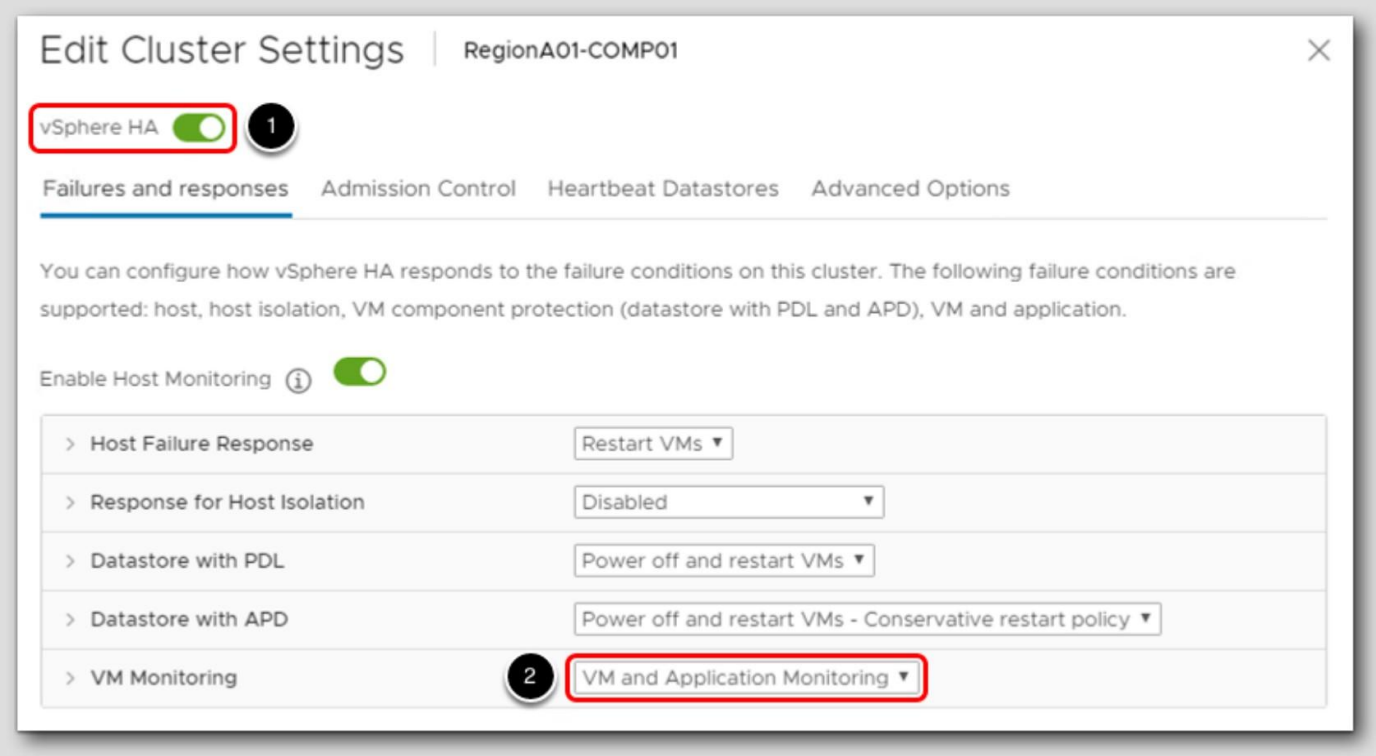


Enable vSphere HA

1. Click the toggle next to vSphere HA to enable it.

2.From the VM Monitoring drop-down list, select VM and Application Monitoring.

By selecting VM and Application Monitoring, a VM will be restarted if heartbeats are not received within a set time, the default is 30 seconds.

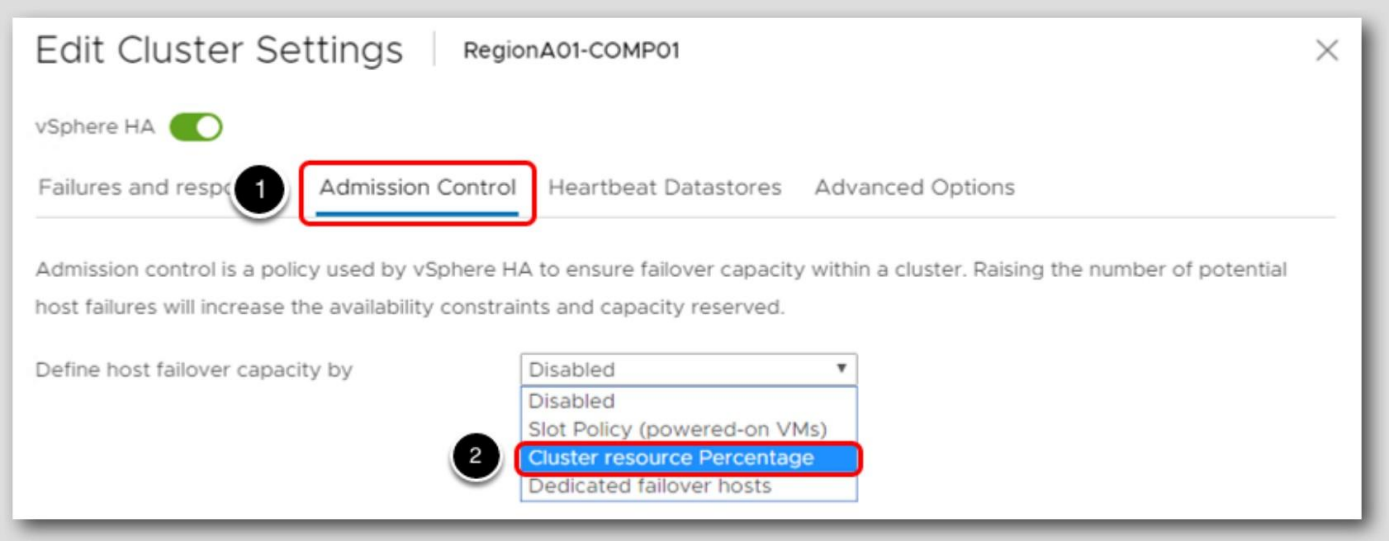


Admission Control

1. Click the Admission Control tab.

2.In the Define host failover capacity by drop-down menu, select Cluster resource Percentage.

We are setting aside a certain percentage of CPU and Memory resources to be used for failover, in the above case 25% for each.



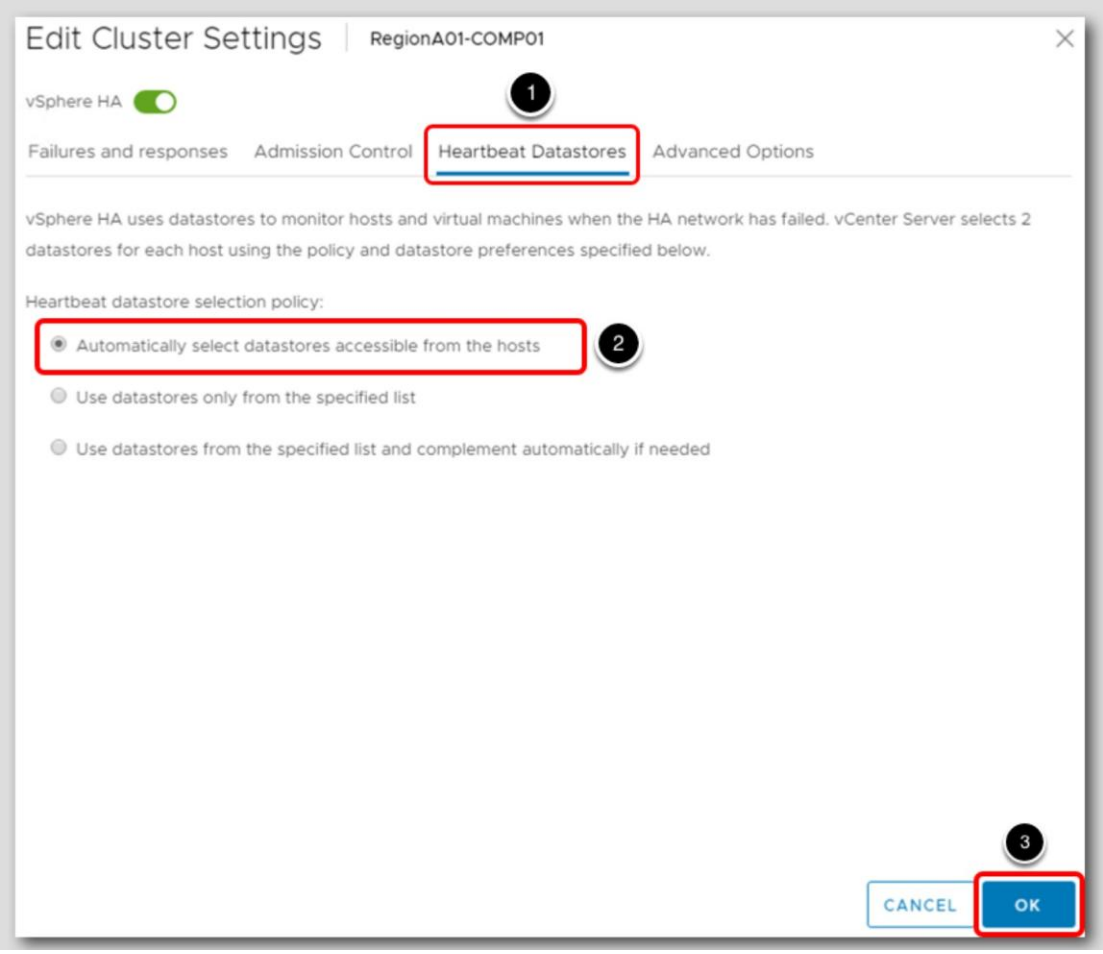
Heartbeat Datastores

1. Click Heartbeat Datastores

2.Select Automatically select datastores accessible from the hosts.

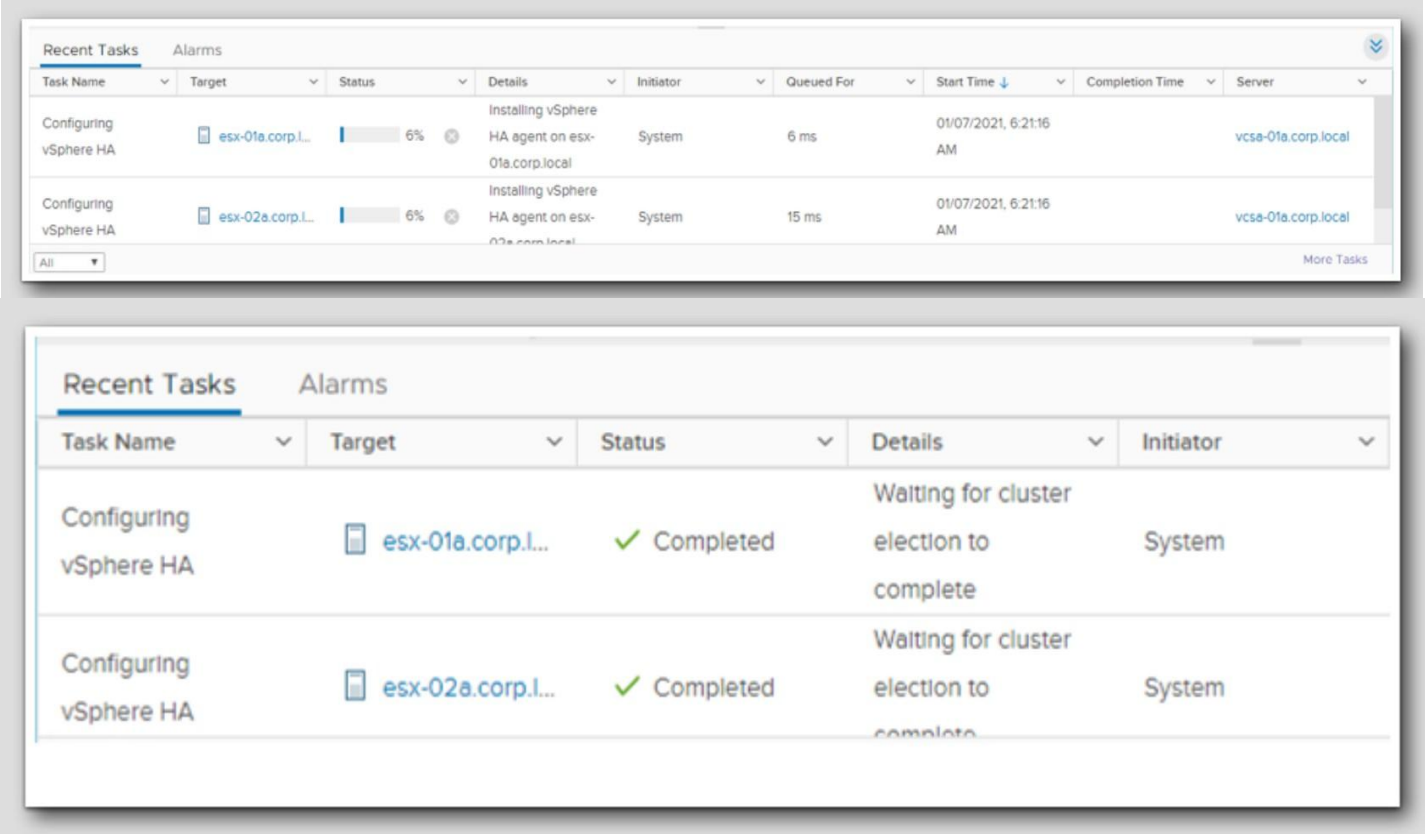
This is another layer of protection. Heartbeat Datastores allows vSphere HA to monitor hosts when a management network partition occurs and to continue to respond to failures that occur.

3. Click OK to enable vSphere HA



Monitor the task

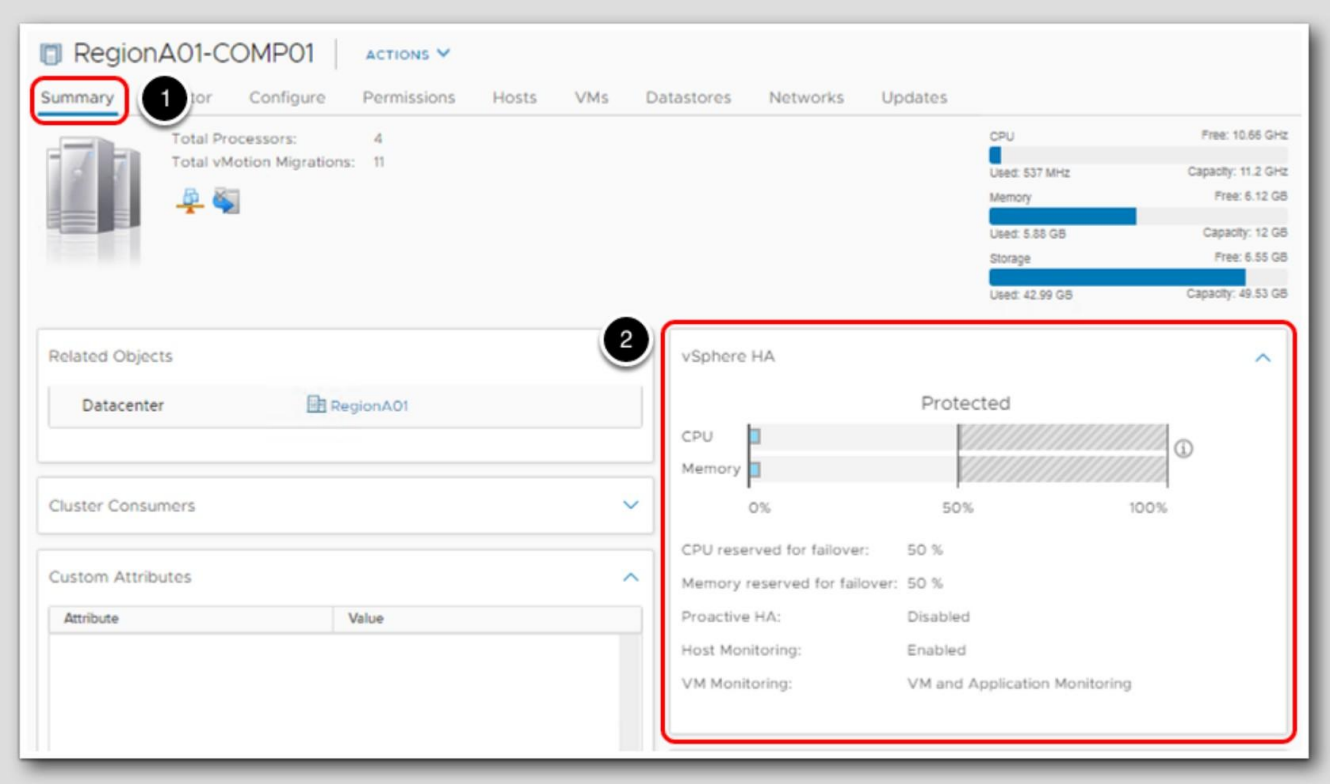
It will take a minute or two to configure vSphere HA. You can monitor the progress in the Recent Tasks window.



Use the Summary Tab to Verify that HA Is Enabled

1. Click the Summary tab

2.Locate and expand the vSphere HA panel in the data area: click on the ">" to the right of the panel's name to expand it.



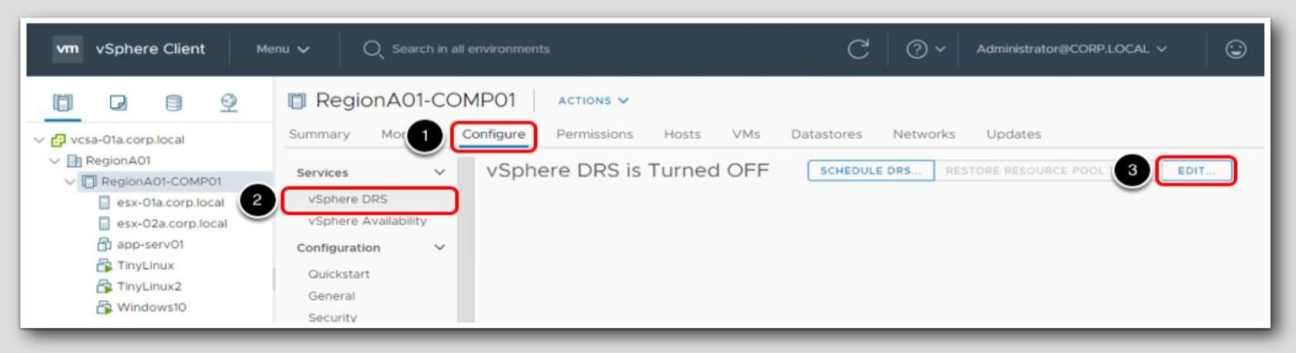
If vSphere HA does not show Protected and the tasks completed successfully, you may need to click the refresh button.

Notice the bars that display resource usage in blue, protected capacity in light gray, and reserve capacity using stripes.

Practical 12: Implement a vSphere DRS cluster.

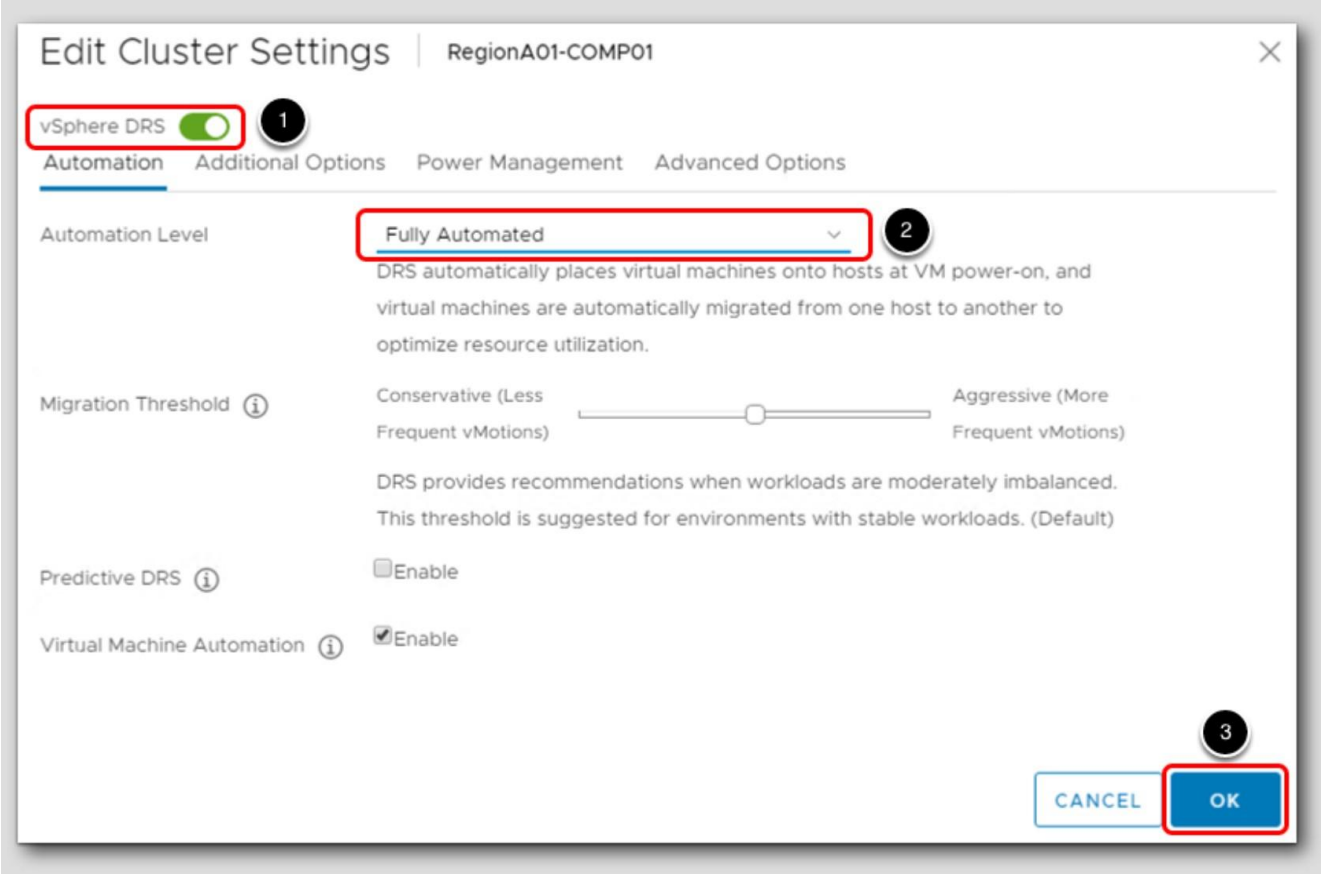
Enable Distributed Resource Scheduler (DRS)

1. Click on the Configure tab to start the process of enabling Distributed Resource Scheduler.
2. Click vSphere DRS.
3. Click on the Edit button to modify the DRS settings.

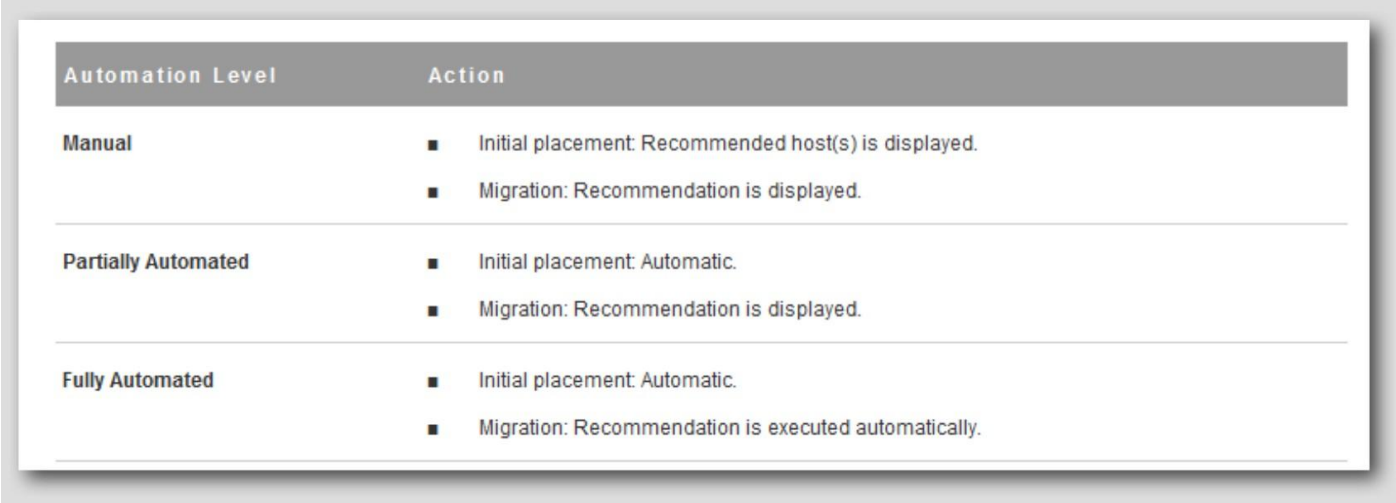


Enable Distributed Resource Scheduler (DRS)

1. Verify that vSphere DRS is enabled. If not, click the vSphere DRS to enable.
2. Click the drop-down box and select Fully Automated.
3. Click OK.



Automation Levels



Automation Level	Action
Manual	■ Initial placement: Recommended host(s) is displayed. ■ Migration: Recommendation is displayed.
Partially Automated	■ Initial placement: Automatic. ■ Migration: Recommendation is displayed.
Fully Automated	■ Initial placement: Automatic. ■ Migration: Recommendation is executed automatically.

The chart shown above is showing how DRS affects placement and migration according to the setting Manual, Partially Automated or Fully Automated.

Use the Cluster's Summary Tab to Check Cluster Balance

1. Click the Summary tab to display the current status of the cluster.
2. The Summary tab of the Cluster RegionA01-COMP01 shows the current balance of the cluster.Also shown in the DRS section is how many recommendations or faults that have occurred with the cluster. (You may have to scroll down to see the vSphere DRS widget)