Analysis of Threats and Attacks on Privacy and Security of Cloud Computing Using Machine Learning and Deep Learning

Importing Necesaary Libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

Importing Dataset

In [3]:
```
df = pd.read_csv('cloud_computing.csv')
```

Data Exploration

In [4]:
```
df.head()
```

Out[4]:

	vm_id	timestamp	cpu_usage	memory_usage	network_traffic	power_consumption	num_executed_instructions	execution_time	en
0	c5215826-6237-4a33-9312-72c1df909881	25-01-2023 09:10	54.881350	78.950861	164.775973	287.808986	7527.0	69.345575	
1	29690bc6-1f34-403b-b509-a1ecb1834fb8	26-01-2023 04:46	71.518937	29.901883	NaN	362.273569	5348.0	41.396040	
2	2e55abc3-5bad-46cb-b445-a577f5e9bf2a	13-01-2023 23:39	NaN	92.709195	203.674847	231.467903	5483.0	24.602549	
3	e672e32f-c134-4fbc-992b-34eb63bef6bf	09-02-2023 11:45	54.488318	88.100960	NaN	195.639954	5876.0	16.456670	
4	f38b8b50-6926-4533-be4f-89ad11624071	14-06-2023 08:27	42.365480	NaN	NaN	359.451537	3361.0	55.307992	

In [5]:
```
df.tail()
```

Out[5]:

	vm_id	timestamp	cpu_usage	memory_usage	network_traffic	power_consumption	num_executed_instructions	execution_tir
1048570	48aae80b-a2f3-4c70-b31c-6bd9ad8077c2	21-04-2023 11:10	32.859886	71.297520	NaN	246.616398	9342.0	71.2585
1048571	0f5c5e91-1196-4380-99ad-0bb54591ef42	12-05-2023 23:50	25.588859	4.065927	712.940448	69.954795	7808.0	79.8973
1048572	b231bafe-5409-425a-9e15-d99dc5939b36	11-03-2023 22:37	53.333198	73.110515	677.152821	182.172797	1456.0	Na
1048573	b9b87b7a-5cbe-4fd5-9d58-7340f8ce6825	08-05-2023 21:12	45.626507	46.779306	NaN	472.255335	NaN	93.6956
1048574	64cd82aa-3546-4575-b6fe-4faa2cc19980	09-02-2023 07:10	12.492005	93.750679	303.787255	29.085574	7434.0	37.5336

In [6]:
```
df.shape
```

Out[6]:
```
(1048575, 12)
```

In [7]:
```
df.columns
```

Out[7]:
```
Index(['vm_id', 'timestamp', 'cpu_usage', 'memory_usage', 'network_traffic',
       'power_consumption', 'num_executed_instructions', 'execution_time',
       'energy_efficiency', 'task_type', 'task_priority', 'task_status'],
      dtype='object')
```

In [8]:
```
df.duplicated().sum()
```

Out[8]:
```
0
```

In [9]:
```
df.isnull().sum()
```

Out[9]:
```
vm_id                       105029
timestamp                   105631
cpu_usage                   104310
memory_usage                105379
network_traffic             104627
power_consumption           105153
num_executed_instructions   104737
execution_time              104987
energy_efficiency           104756
task_type                   104979
task_priority               104948
task_status                 105281
dtype: int64
```

In [10]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1048575 entries, 0 to 1048574
Data columns (total 12 columns):
 #   Column                     Non-Null Count   Dtype  
---  ------                     --------------   -----  
 0   vm_id                      943546 non-null  object 
 1   timestamp                  942944 non-null  object 
 2   cpu_usage                  944265 non-null  float64
 3   memory_usage               943196 non-null  float64
 4   network_traffic            943948 non-null  float64
 5   power_consumption          943422 non-null  float64
 6   num_executed_instructions  943838 non-null  float64
 7   execution_time             943588 non-null  float64
 8   energy_efficiency          943819 non-null  float64
 9   task_type                  943596 non-null  object 
 10  task_priority              943627 non-null  object 
 11  task_status                943294 non-null  object 
dtypes: float64(7), object(5)
memory usage: 96.0+ MB
```

In [11]:
```
df.describe()
```

Out[11]:

	cpu_usage	memory_usage	network_traffic	power_consumption	num_executed_instructions	execution_time	energy_efficiency
count	944265.000000	943196.000000	943948.000000	943422.000000	943838.000000	943588.000000	9.438190e+05
mean	50.048163	49.963042	500.065179	250.227086	5001.324844	50.001421	5.000876e-01
std	28.873220	28.840676	288.677027	144.329154	2885.381367	28.862121	2.886819e-01
min	0.000071	0.000021	0.000189	0.000192	0.000000	0.000001	1.260000e-07
25%	25.018858	24.998474	250.042065	125.245967	2502.000000	24.974178	2.498607e-01
50%	50.108829	49.947326	500.090079	250.608946	5002.000000	50.027097	5.004590e-01
75%	75.100072	74.923761	750.025461	375.266989	7502.000000	75.008272	7.501253e-01
max	99.999972	99.999928	999.999837	499.999325	9999.000000	99.999956	9.999992e-01

In [12]:
```
df.nunique()
```

Out[12]:
```
vm_id                      943546
timestamp                  277649
cpu_usage                  944226
memory_usage               943160
network_traffic            943906
power_consumption          943336
num_executed_instructions   10000
execution_time             943562
energy_efficiency          943333
task_type                       3
task_priority                   3
task_status                     3
dtype: int64
```

Handling Missing Values

In [13]:
```
missing_percentage = (df.isnull().sum() / len(df)) * 100
```

In [14]:
```
missing_info = pd.DataFrame({'Column': df.columns, 'Missing Percentage': missing_percentage})
```

In [15]:
```
missing_info = missing_info.sort_values(by='Missing Percentage', ascending=False)
```

In [16]:
```
missing_info
```

Out[16]:

	Column	Missing Percentage
timestamp	timestamp	10.073767
memory_usage	memory_usage	10.049734
task_status	task_status	10.040388
power_consumption	power_consumption	10.028181
vm_id	vm_id	10.016356
execution_time	execution_time	10.012350
task_type	task_type	10.011587
task_priority	task_priority	10.008631
energy_efficiency	energy_efficiency	9.990320
num_executed_instructions	num_executed_instructions	9.988508
network_traffic	network_traffic	9.978018
cpu_usage	cpu_usage	9.947786

In [17]:
```
categorical_columns = ['task_type', 'task_priority', 'task_status']
```

In [18]:
```
for col in categorical_columns:
    mode_value = df[col].mode()[0]
    df[col] = df[col].fillna(mode_value)
```

In [19]:
```
numerical_columns = ['cpu_usage', 'memory_usage', 'network_traffic', 'power_consumption',
                     'num_executed_instructions', 'execution_time', 'energy_efficiency']
```

In the case of nominal data, we use mode. For ordinal data, the median is recommended. Mean is widely used to find the central tendency of ratioed / interval variables. But the mean is not always the right choice to determine the central tendency because if the dataset contains outliers, the mean will be very high or low. In that case, the median is more robust than the mean. We will use the median if the median is greater or less than the mean. Otherwise, mean is the best choice.

In [20]:
```
mean_values = df[numerical_columns].mean()
median_values = df[numerical_columns].median()
```

In [21]:
```
mean_values
```

Out[21]:
```
cpu_usage                     50.048163
memory_usage                  49.963042
network_traffic              500.065179
power_consumption            250.227086
num_executed_instructions   5001.324844
execution_time                50.001421
energy_efficiency              0.500088
dtype: float64
```

In [22]:
```
median_values
```

Out[22]:
```
cpu_usage                     50.108829
memory_usage                  49.947326
network_traffic              500.090079
power_consumption            250.608946
num_executed_instructions   5002.000000
execution_time                50.027097
energy_efficiency              0.500459
dtype: float64
```

In [23]:
```
threshold = 0.1
use_median = (abs(median_values - mean_values) > threshold)
```

In [24]:
```
use_median
```

Out[24]:
```
cpu_usage                   False
memory_usage                False
network_traffic             False
power_consumption            True
num_executed_instructions    True
execution_time              False
energy_efficiency           False
dtype: bool
```

In [25]:
```
for col in numerical_columns:
    if use_median[col]:
        df[col] = df[col].fillna(median_values[col])
    else:
        df[col] = df[col].fillna(mean_values[col])
```

In [26]:
```
df['timestamp'] = pd.to_datetime(df['timestamp'])
```

pd.to_datetime() is a function provided by the pandas library in Python, and it is used to convert input data into datetime objects. This

function is particularly useful when dealing with date and time data in various formats.

The pd.to_datetime() function can handle a wide range of input types, including strings, integers, and floating-point numbers, and convert them into pandas datetime objects.

In [27]:
```
df['timestamp'] = df['timestamp'].fillna(method='ffill')
```

method='ffill': The method parameter specifies the filling method to be used. In this case, 'ffill' stands for "forward fill." It means that the missing values will be filled with the previous (i.e., the most recent) non-missing value in the column. This method is often used when dealing with time series data or data where missing values should be carried forward in a logical manner.

In [28]:
```
min_date = df['timestamp'].min()
max_date = df['timestamp'].max()
```

In [29]:
```
min_date
```

Out[29]:
```
Timestamp('2023-01-01 00:00:00')
```

In [30]:
```
max_date
```

Out[30]:
```
Timestamp('2023-12-07 23:59:00')
```

In [31]:
```
df = df.sort_values(by='timestamp')
```

In [32]:
```
df
```

Out[32]:

	vm_id	timestamp	cpu_usage	memory_usage	network_traffic	power_consumption	num_executed_instructions	execution_tim
454772	17fa09d5-4039-4adf-b8c2-92eb116d940a	2023-01-01 00:00:00	50.048163	90.540824	845.326434	35.834837	6556.0	33.23828
898414	fe5e04e3-4ea4-48cf-a193-3f58abd147ed	2023-01-01 00:00:00	14.019569	74.367251	600.520958	368.735533	8240.0	50.74239
969845	2206b2b6-a41c-4755-93d9-2a1784ba66a9	2023-01-01 00:00:00	29.608038	58.788456	345.979118	202.838902	8774.0	40.73441
969844	NaN	2023-01-01 00:00:00	30.719806	89.199261	604.263594	325.820755	5002.0	3.23236
454771	a5ee77f0-5a60-474d-8a66-11bf38337326	2023-01-01 00:00:00	63.282078	49.963042	212.708467	358.127312	3983.0	45.30299
...	...	...	...	...	...	...	...	.
536024	2983b165-6811-4d50-bf63-f2e0bebd893e	2023-12-07 23:59:00	80.090971	47.878890	710.638233	183.238495	3821.0	11.91968
579029	a8120feb-7766-4356-b50c-22c67dd39deb	2023-12-07 23:59:00	30.351574	75.188533	70.264991	249.273443	5002.0	86.87770
393031	f9aa933d-e9a2-472a-8d2b-2aa3e85a22f0	2023-12-07 23:59:00	58.879801	91.365817	47.137432	156.109213	5473.0	57.10734
393030	9b4430c3-15cf-4a36-92c9-bc247b4efa21	2023-12-07 23:59:00	62.949990	43.526760	828.012861	190.164149	4490.0	74.07574
405795	10eac306-4e22-462c-af37-96a4789cc258	2023-12-07 23:59:00	95.447495	93.224728	266.471922	402.789075	6526.0	61.58673

1048575 rows × 12 columns

Data Visualization

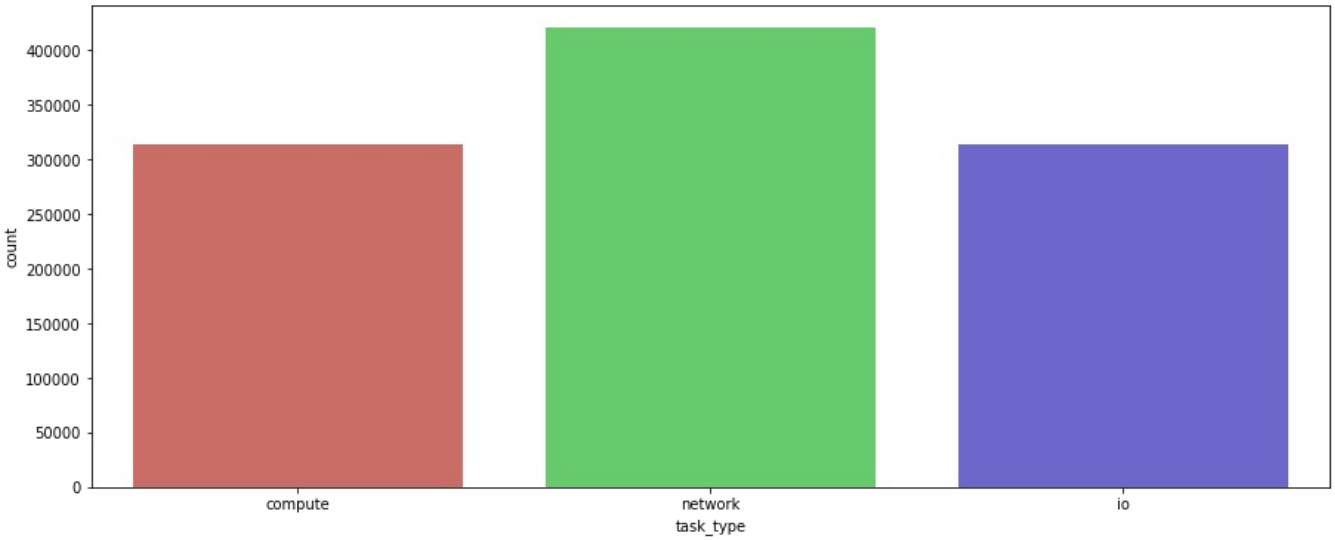
In [33]:
```
for i in categorical_columns:
    print(i,':')
    print(df[i].unique())
    print('\n')
```

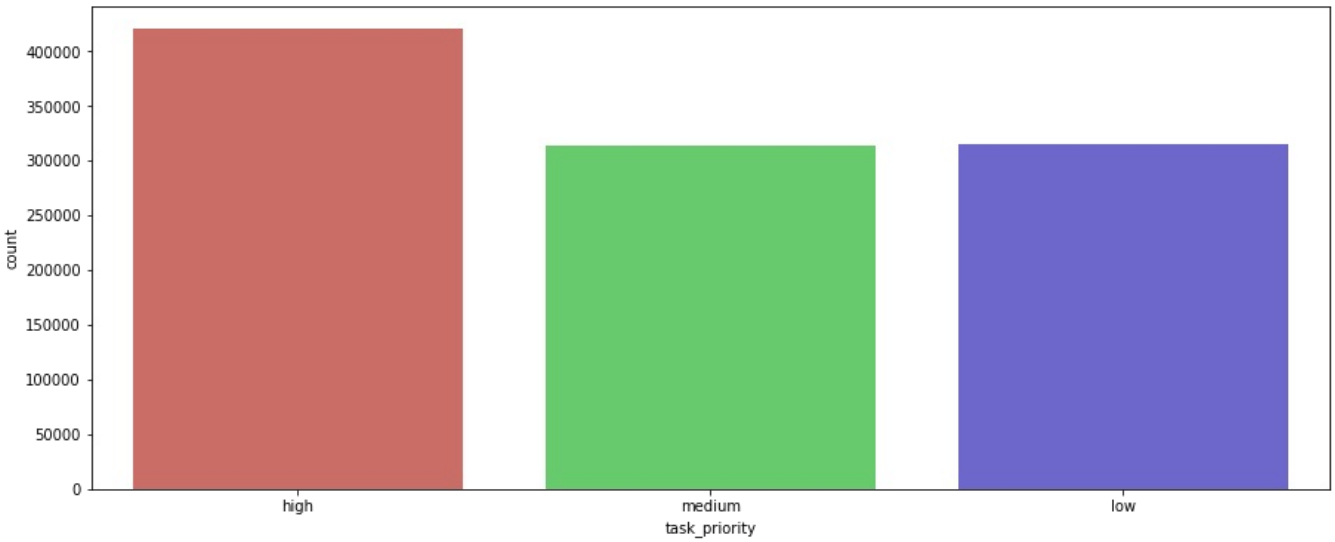
```
task_type :
['compute' 'network' 'io']


task_priority :
['high' 'medium' 'low']


task_status :
['completed' 'waiting' 'running']
```

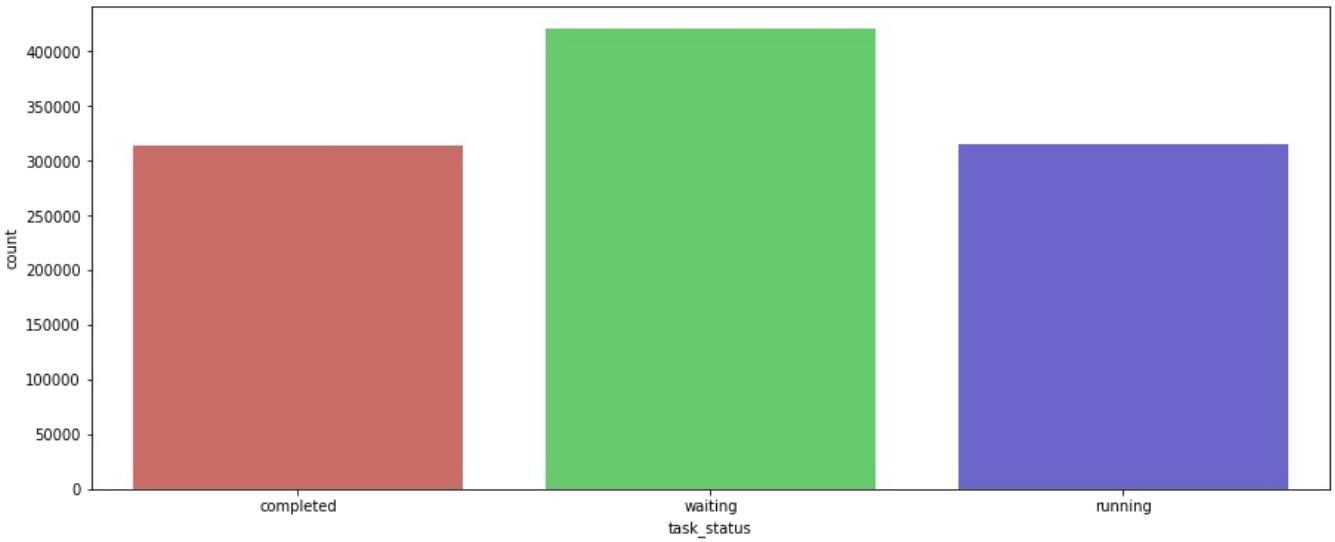
In [34]:
```
for i in categorical_columns:
    print(i,':')
    print(df[i].value_counts())
    print('\n')
```

```
task_type :
network    420620
io         314157
compute    313798
Name: task_type, dtype: int64


task_priority :
high      420002
low       314416
medium    314157
Name: task_priority, dtype: int64


task_status :
waiting      420163
running      314373
completed    314039
Name: task_status, dtype: int64
```

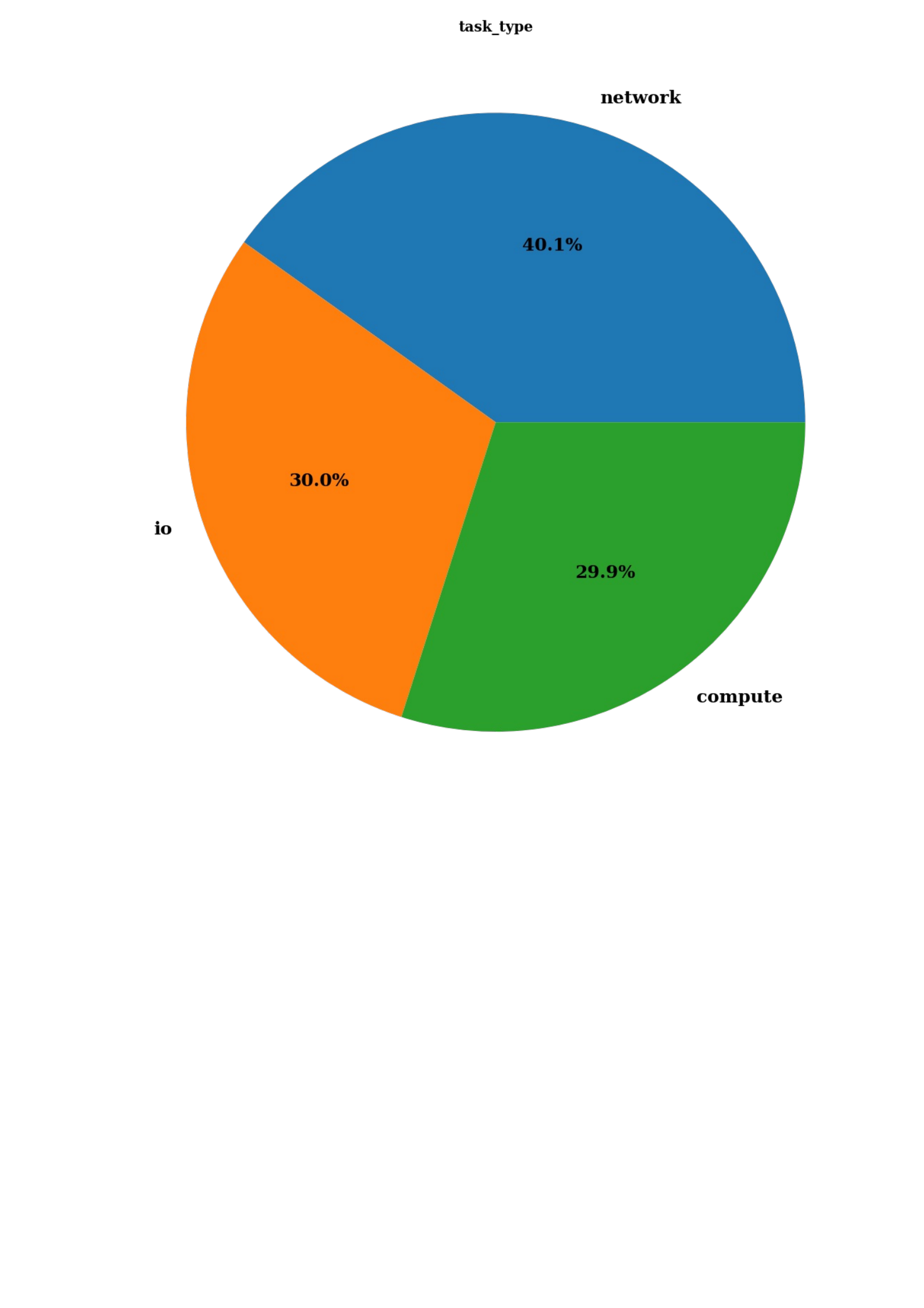
In [35]:
```
for i in categorical_columns:
    plt.figure(figsize=(15,6))
    sns.countplot(x = df[i], data = df, palette = 'hls')
    plt.show()
```

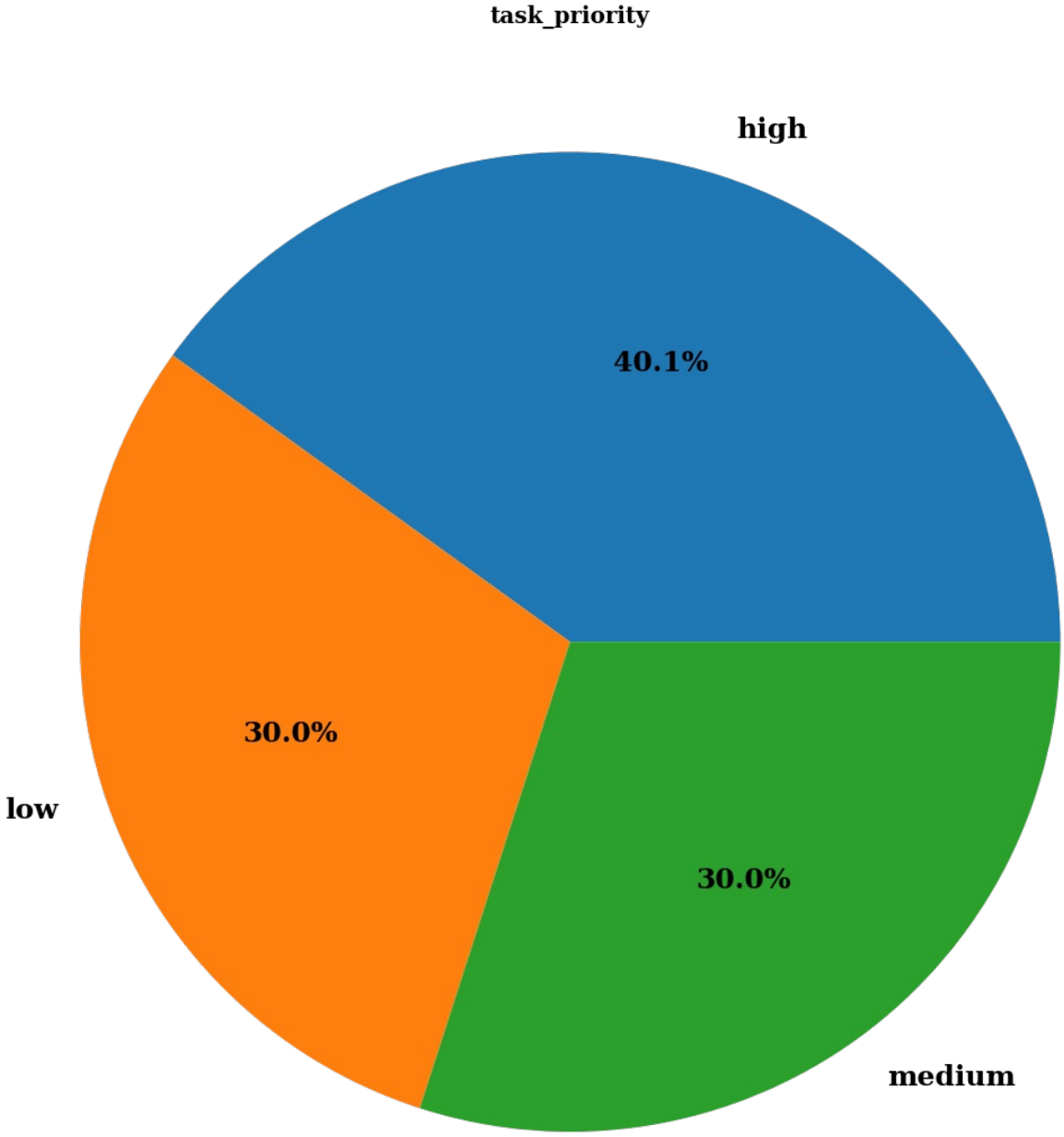


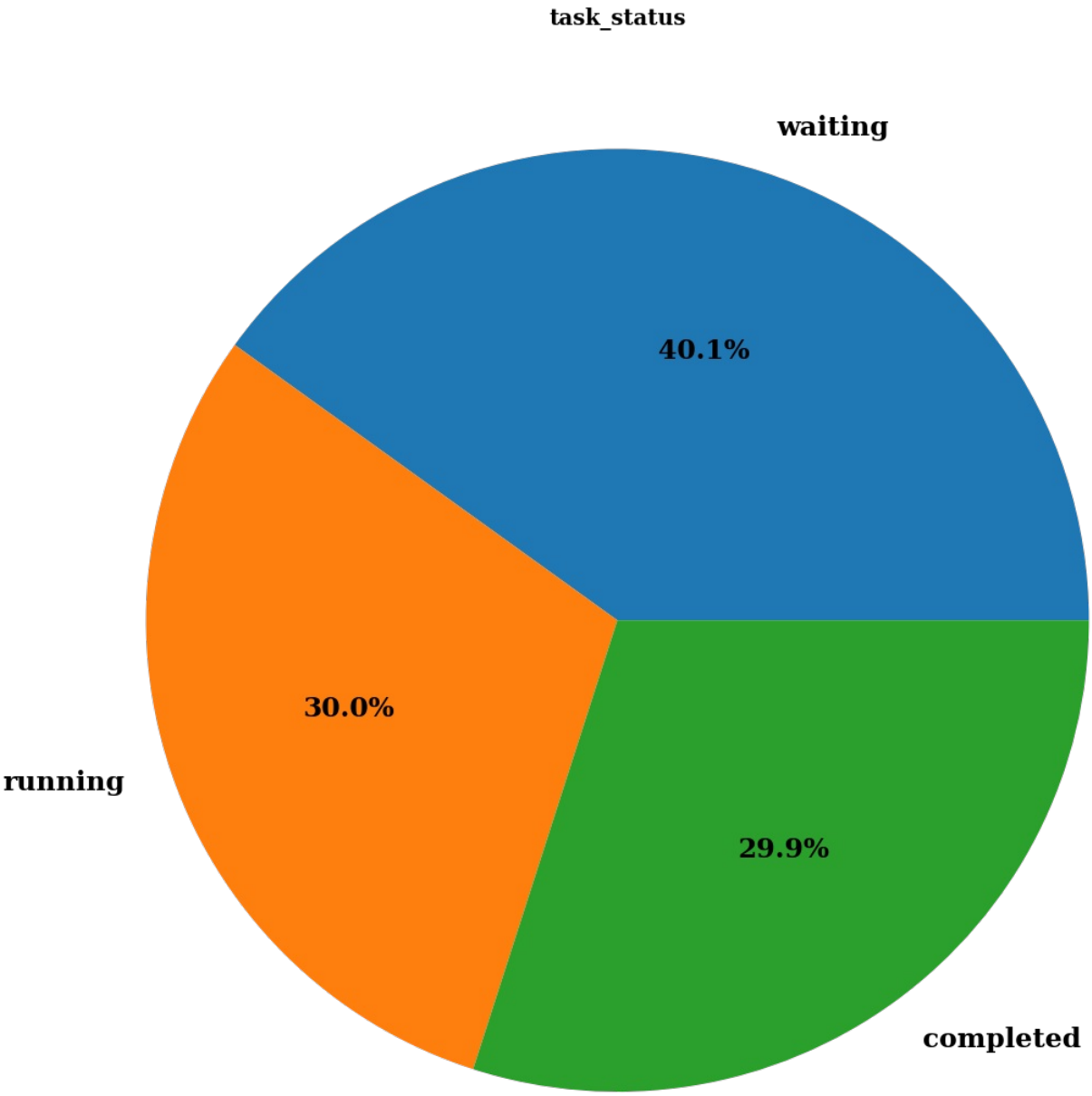




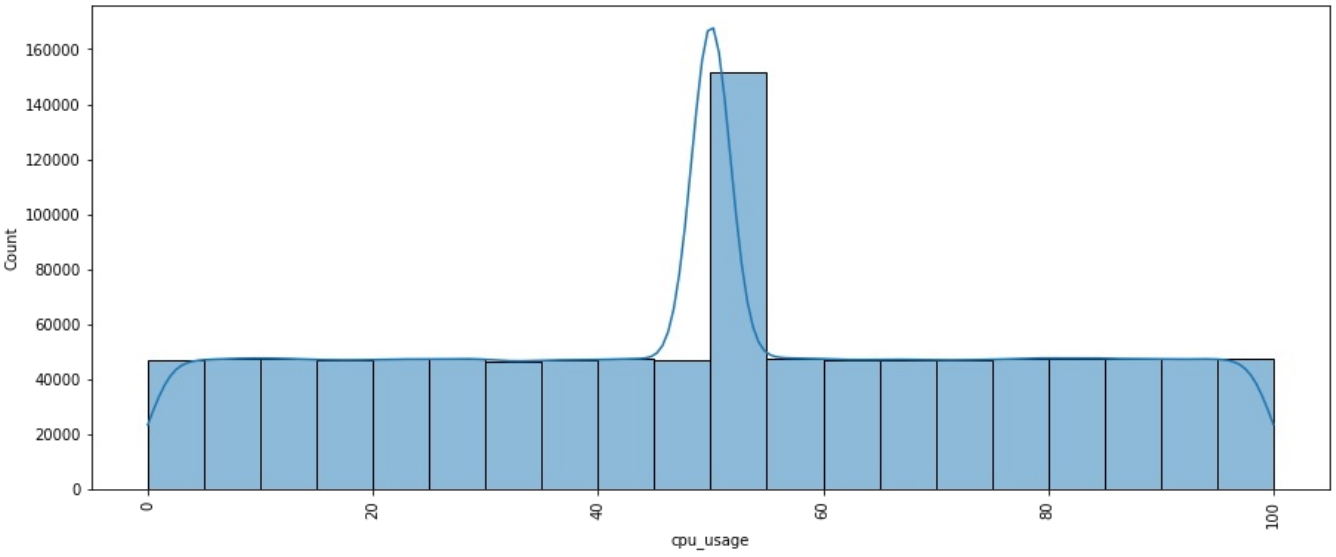
In [36]:
```
for i in categorical_columns:
    plt.figure(figsize=(30,20))
    plt.pie(df[i].value_counts(), labels=df[i].value_counts().index, autopct='%1.1f%%', textprops={ 'fontsize':
                                        'color': 'black',
                                        'weight': 'bold',
                                        'family': 'serif' })
    hfont = {'fontname':'serif', 'weight': 'bold'}
    plt.title(i, size=20, **hfont)
    plt.show()
```

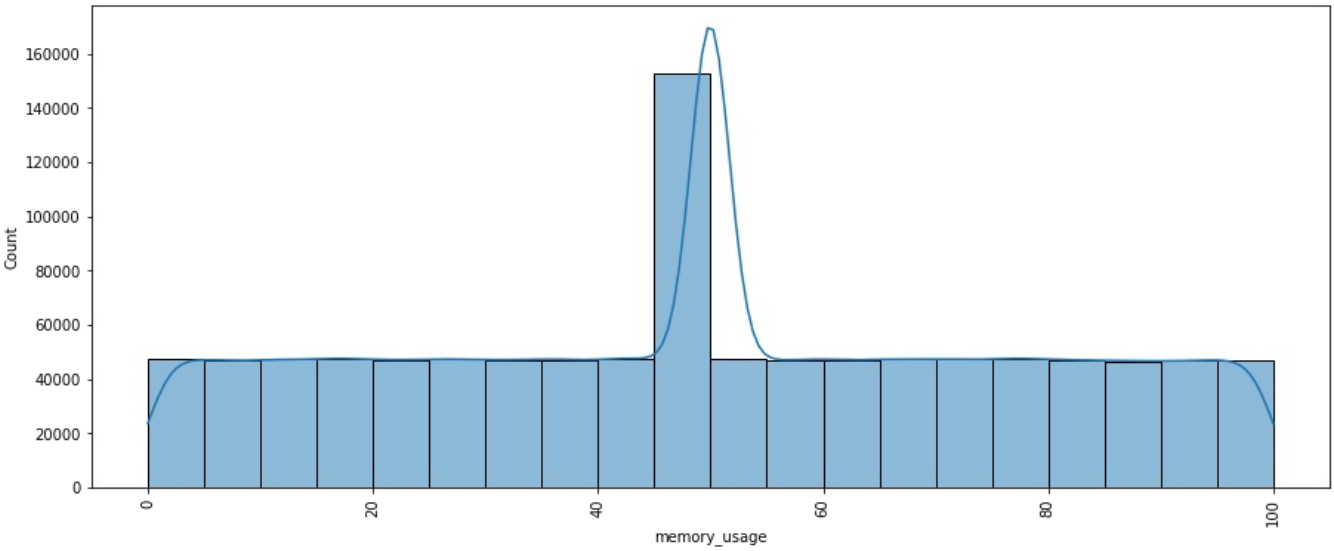

task_type
network
40.1%
30.0%
io
29.9%
compute

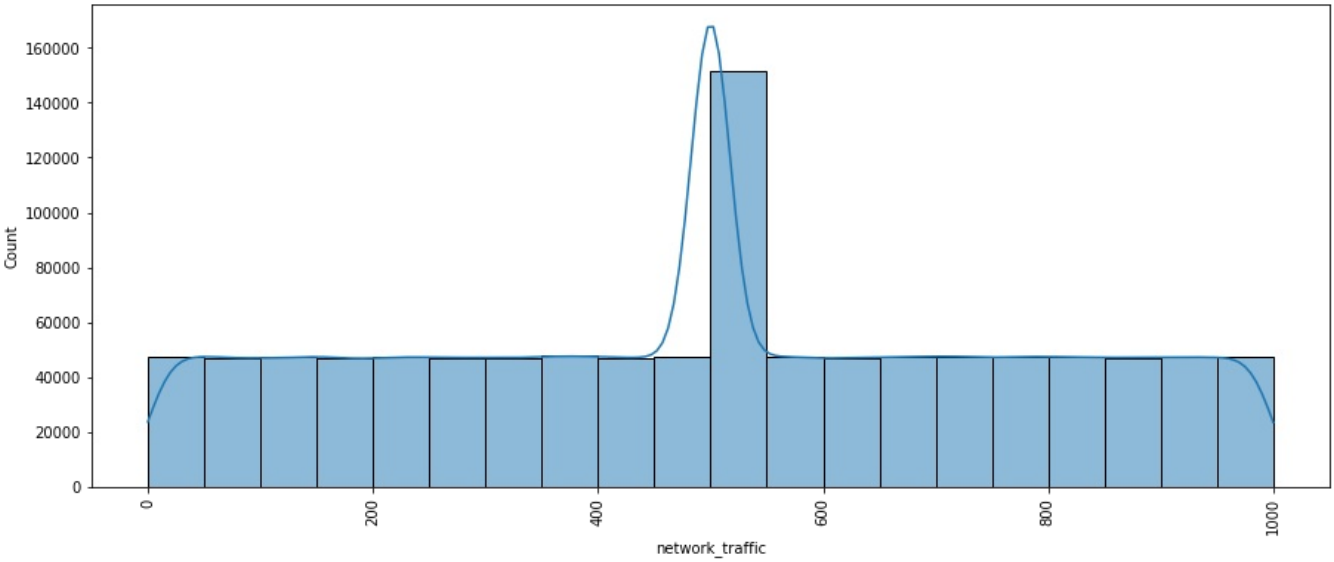

task_priority
high
40.1%
30.0%
low
30.0%
medium

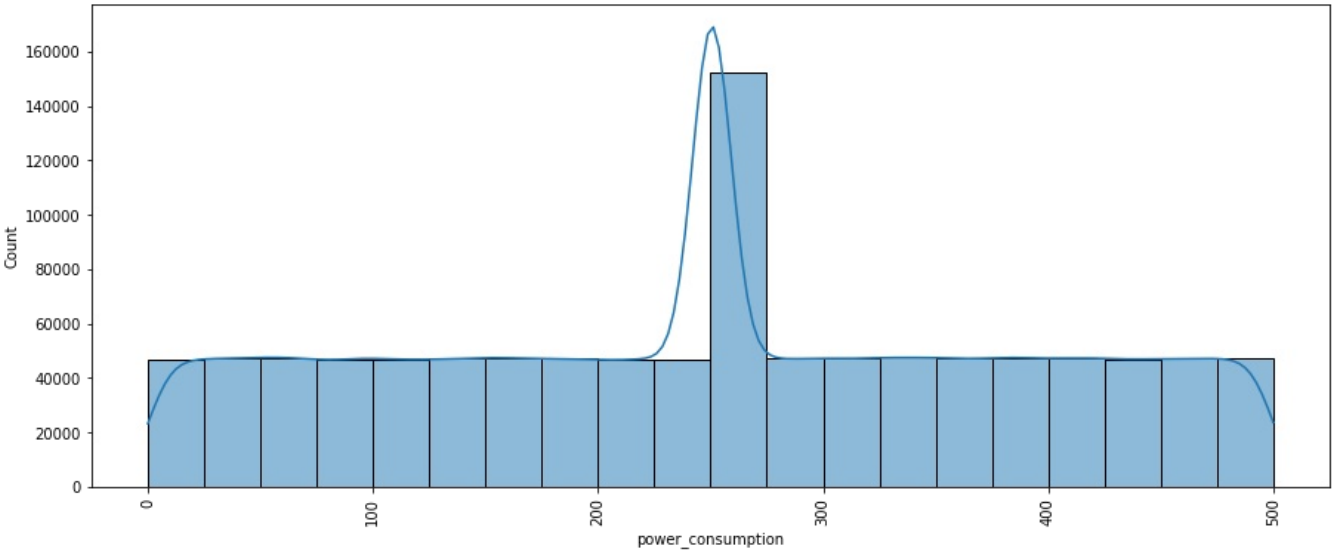


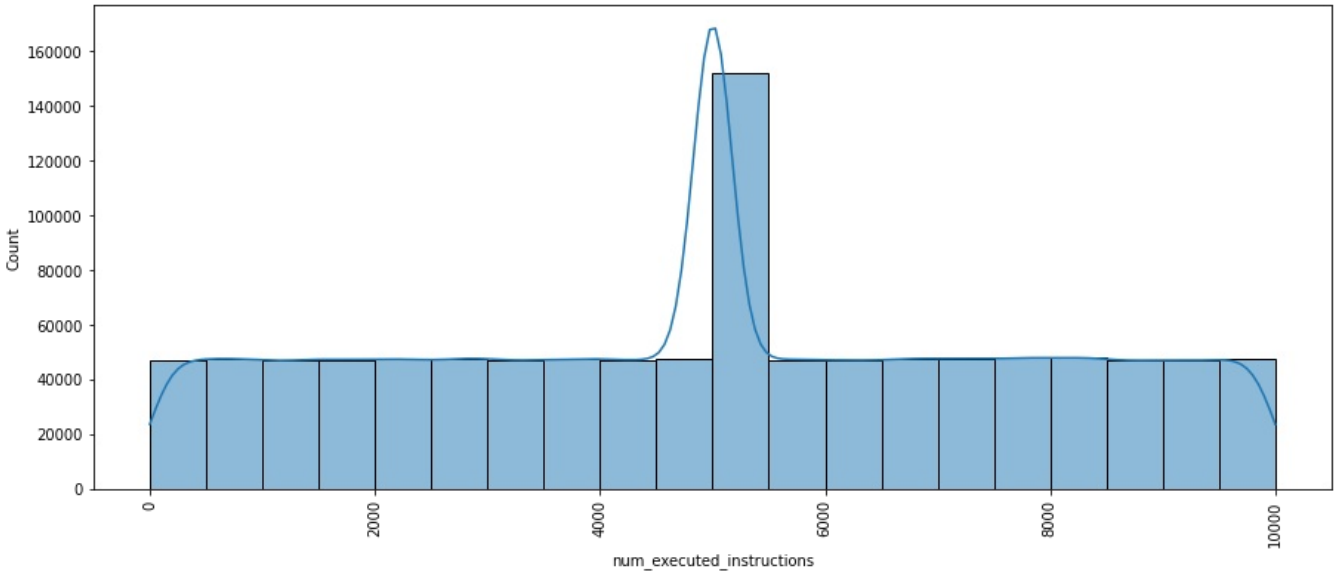
In [37]:
```
for i in numerical_columns:
    plt.figure(figsize=(15,6))
    sns.histplot(df[i], kde = True, bins = 20, palette = 'hls')
    plt.xticks(rotation = 90)
    plt.show()
```

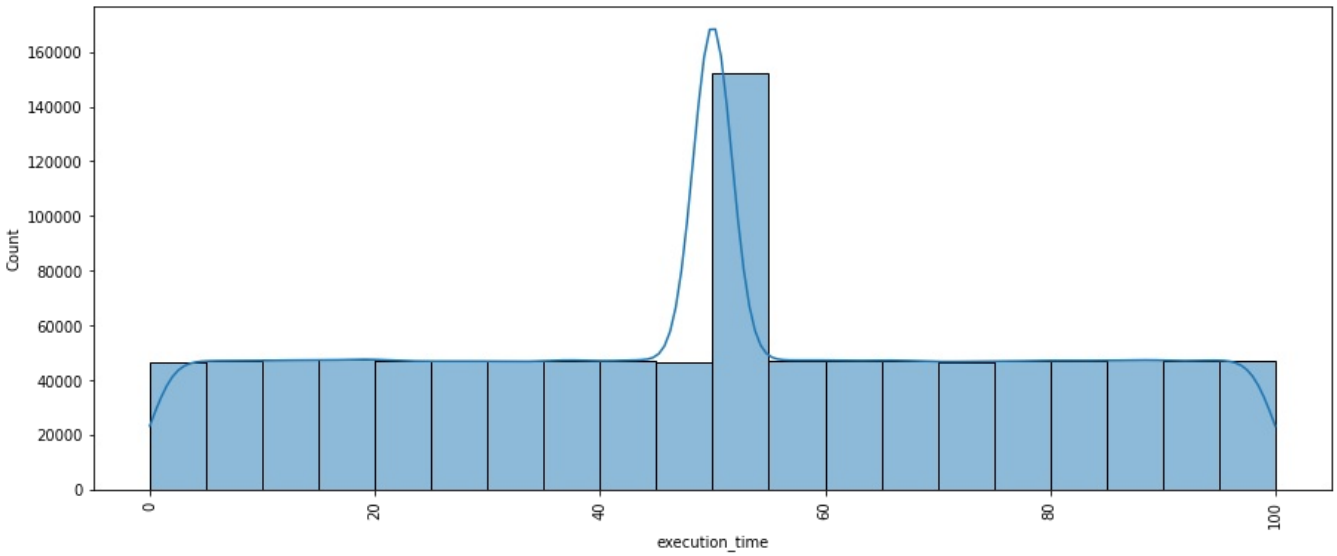


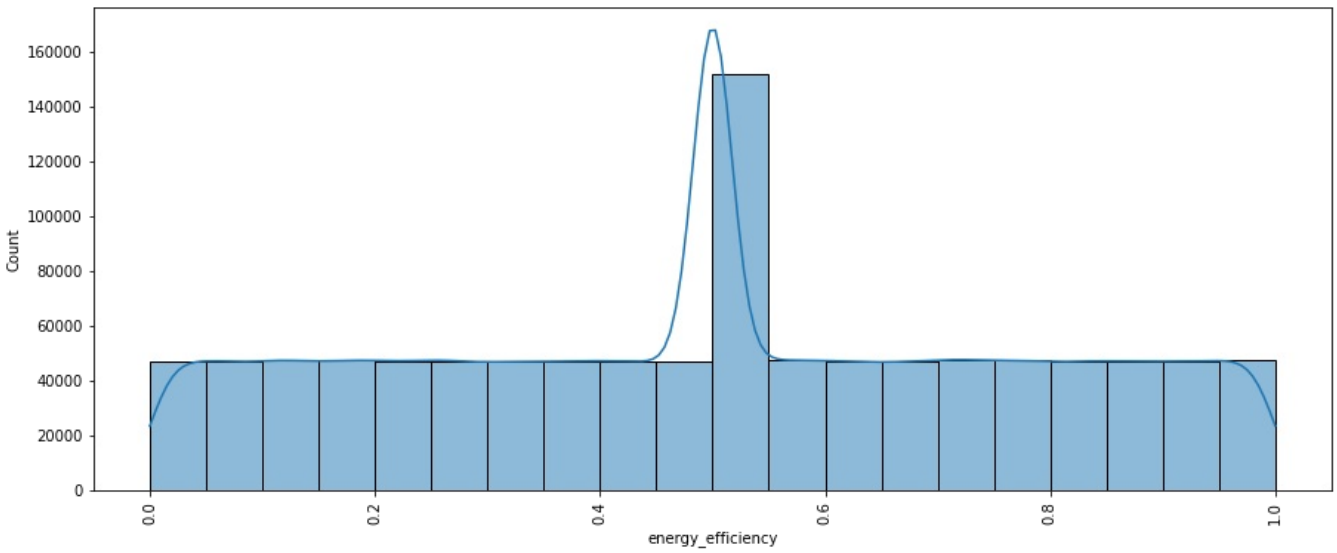

Count
160000
140000
120000
100000
80000
60000
40000
20000
0
0
20
40
60
80
100
memory_usage


Count
160000
140000
120000
100000
80000
60000
40000
20000
0
0
200
400
600
800
1000
network_traffic


Count
160000
140000
120000
100000
80000
60000
40000
20000
0
0
100
200
300
400
500
power_consumption







KDE can have various meanings depending on the context, but in the realm of data analysis, it most commonly refers to **Kernel Density Estimation**. Kernel Density Estimation is a non-parametric way to estimate the probability density function of a random variable. It is a

method for visualizing the distribution of data, especially in the case of continuous data.

Here's a basic explanation of Kernel Density Estimation (KDE):

- **Probability Density Function (PDF):** In statistics, a probability density function describes the likelihood of a random variable taking on a specific value. For continuous data, the PDF represents the distribution of the data as a continuous curve.
- **Kernel:** In the context of KDE, a kernel is a smooth, non-negative function. It's often a Gaussian (normal) distribution, but other kernel functions, like the Epanechnikov or triangular kernel, can be used.
- **Estimation:** KDE is a technique that estimates the PDF of a dataset by placing a kernel on each data point and summing or integrating them to produce a smooth, continuous curve. The width or bandwidth of the kernel determines how "smooth" the resulting curve will be. A smaller bandwidth produces a more detailed curve, while a larger bandwidth results in a smoother, less detailed curve.

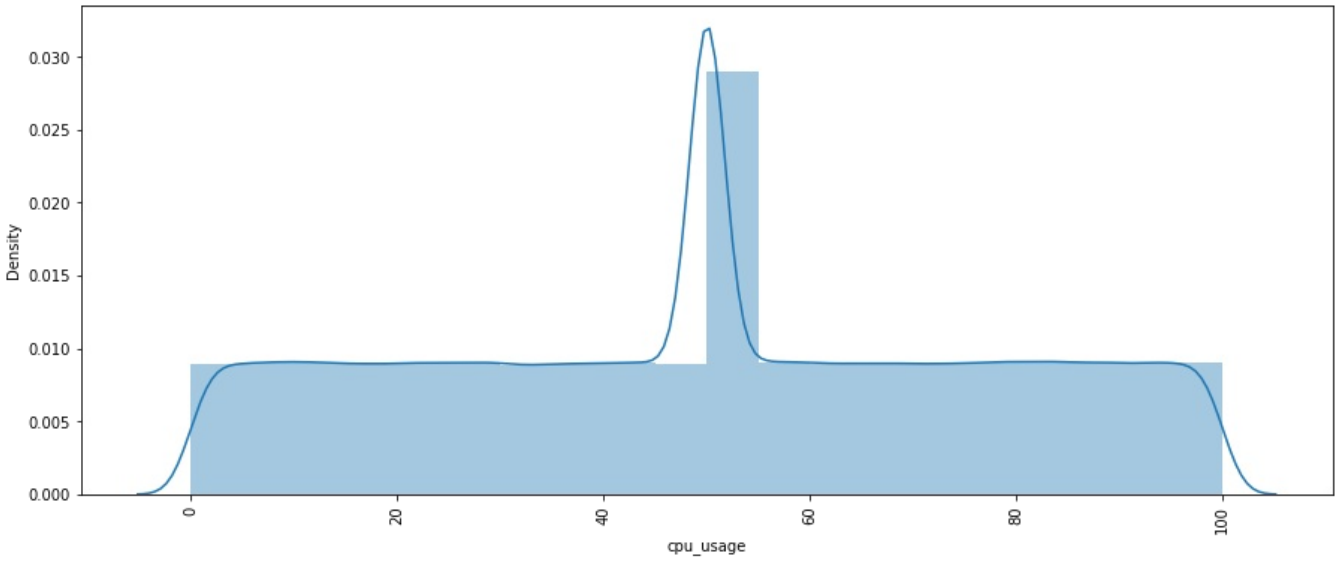
KDE is often used for data visualization to understand the underlying data distribution. It can be particularly useful for:

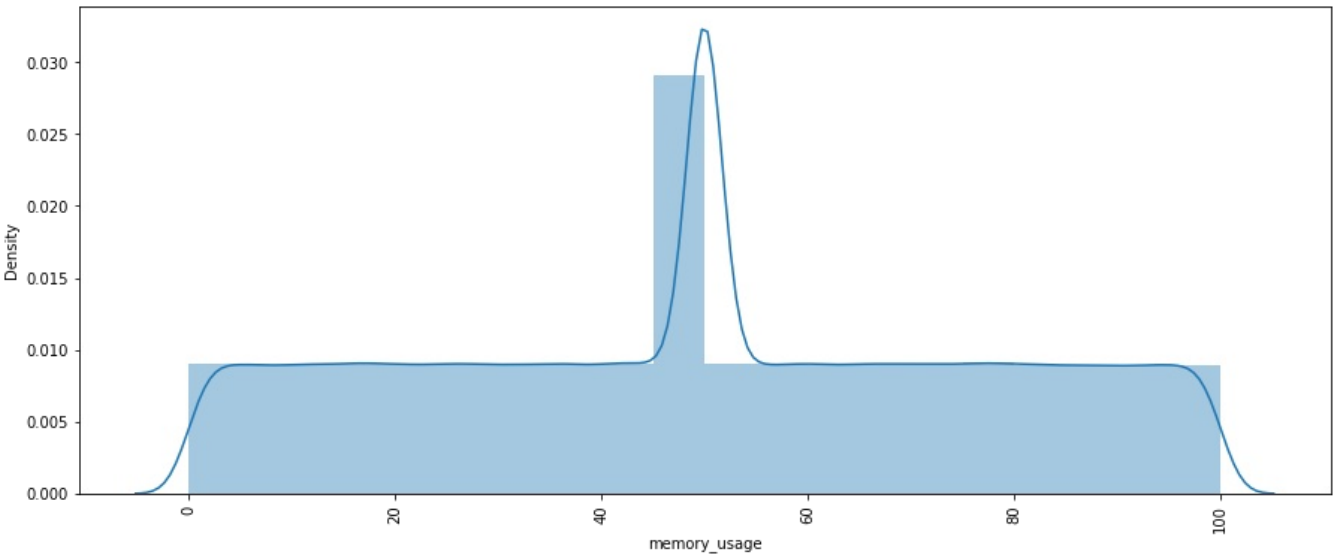
1. **Data Exploration:** KDE provides a way to visualize the shape and characteristics of a dataset, helping you identify patterns and trends.
2. **Comparison:** It can be used to compare the distributions of different datasets.
3. **Density Estimation:** By estimating the PDF, KDE can provide insights into the likelihood of observing data at different points along the distribution.
4. **Anomaly Detection:** Outliers or rare events in a dataset may be easier to spot with KDE.
5. **Smoothing Data:** KDE can be used to smooth noisy data, making it easier to identify underlying patterns.

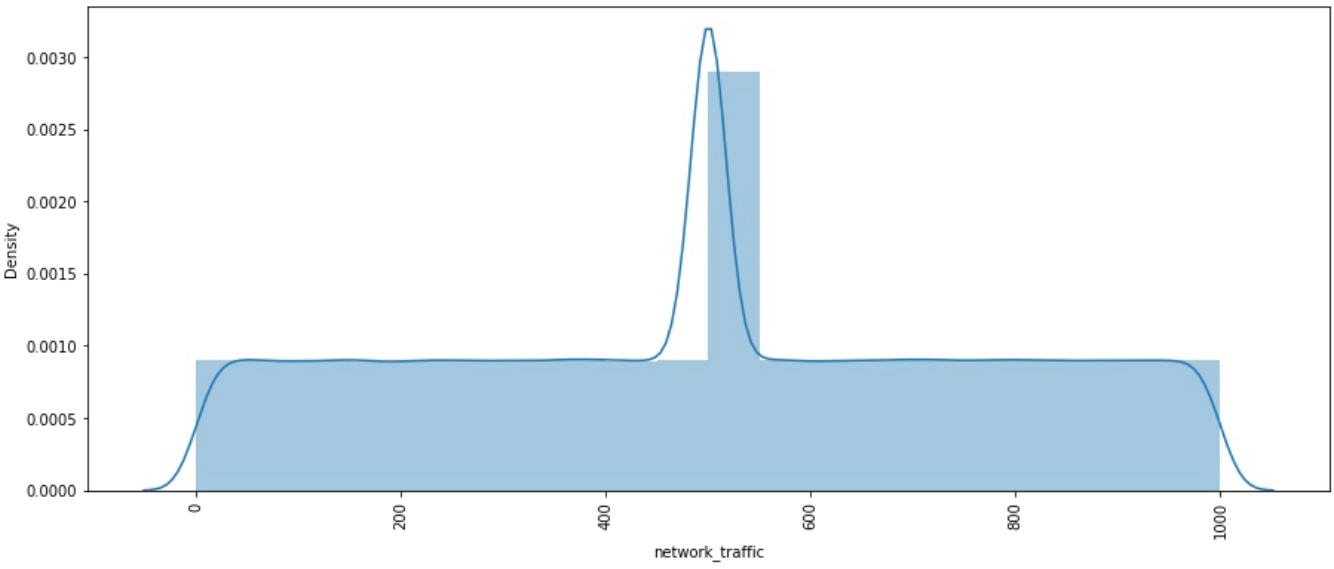
Here's a simple example: If you have a set of data points representing the heights of individuals, you can use KDE to estimate the probability density function of heights in the population. The resulting curve would provide insights into the likelihood of encountering people of different heights.

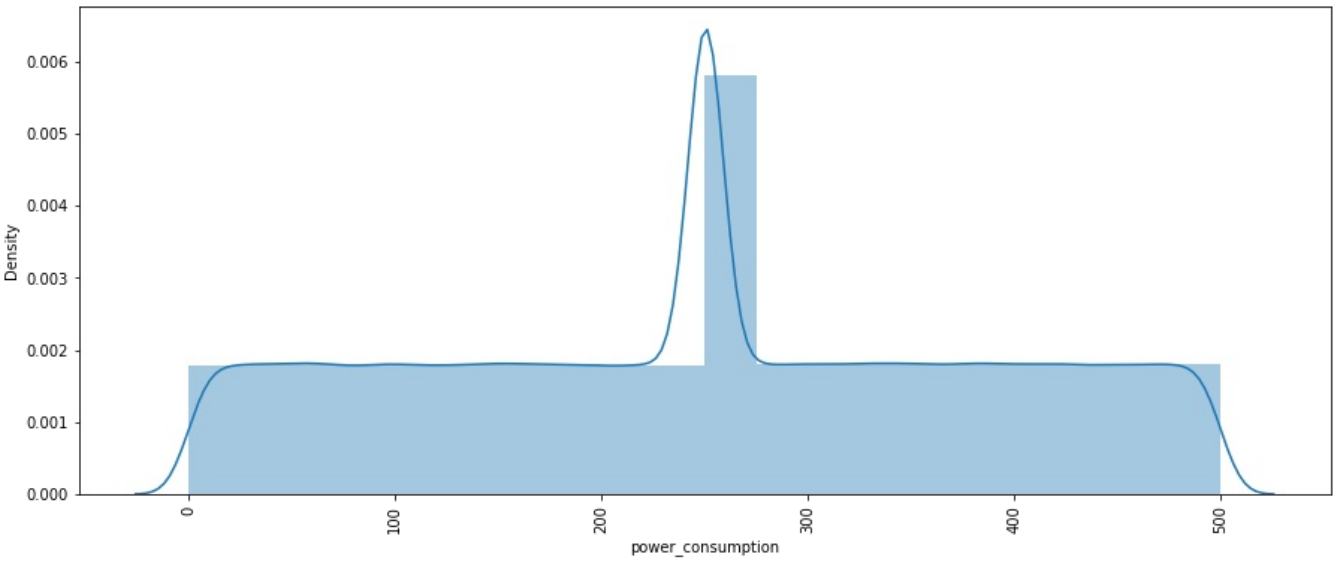
KDE is widely used in various fields, including statistics, data science, machine learning, and data visualization, as it helps to gain a better understanding of data distributions, which is crucial for making informed decisions and drawing meaningful insights from data.

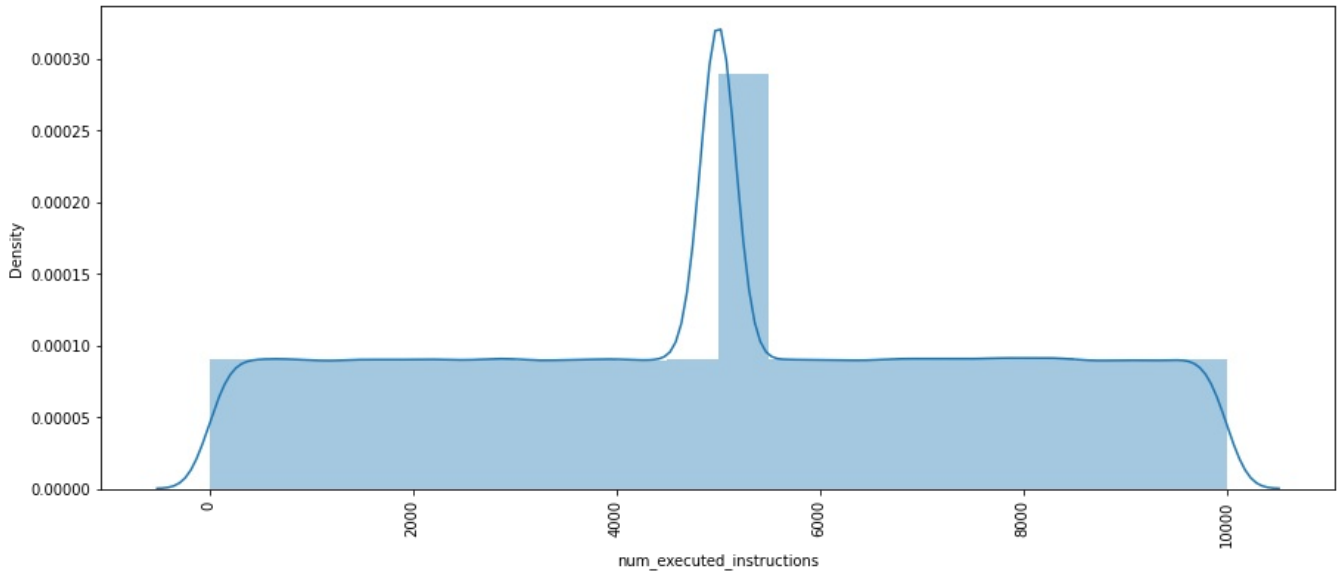
In [38]:
```
for i in numerical_columns:
    plt.figure(figsize=(15,6))
    sns.distplot(df[i], kde = True, bins = 20)
    plt.xticks(rotation = 90)
    plt.show()
```

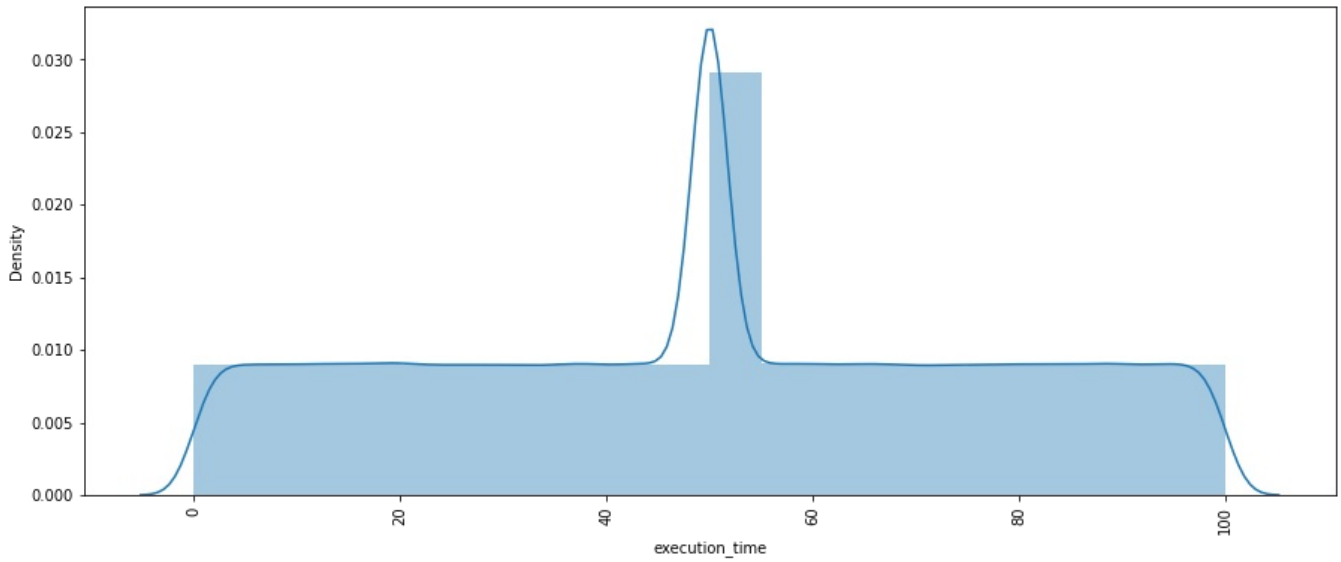


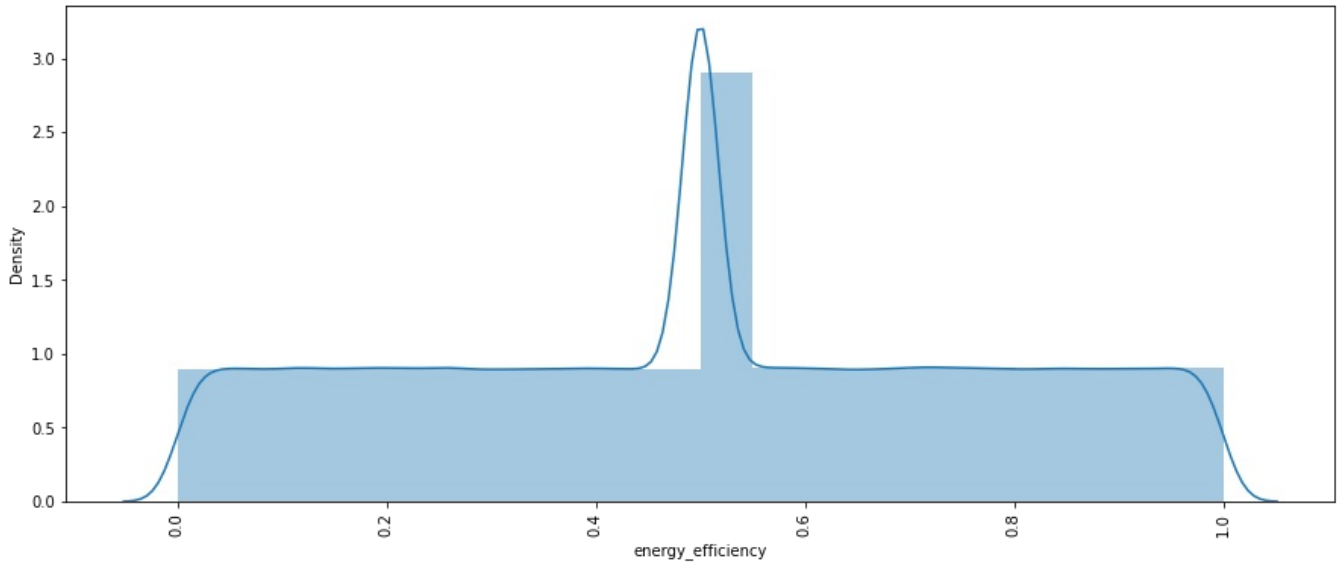

Density
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
20
40
60
80
100
memory_usage


Density
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
200
400
600
800
1000
network_traffic


Density
0.006
0.005
0.004
0.003
0.002
0.001
0.000
0
100
200
300
400
500
power_consumption







A histogram and a distplot (distribution plot) are both used for visualizing the distribution of a dataset. However, they differ in terms of their usage, components, and the additional features they offer. Let's explore the key differences between a histogram and a distplot:

Histogram:

1. **Visualization Method:** A histogram is a graphical representation of the distribution of a dataset. It consists of a series of adjacent bars that represent the frequency or count of data points falling within specified bins or intervals.
2. **Components:** A histogram mainly consists of bars that indicate how many data points fall into each bin. It provides a visual summary of the data's distribution but does not include additional components like a curve or a probability density estimate.
3. **Usage:** Histograms are often used for a basic, no-frills visualization of data distribution. They are well-suited for showing the overall

shape of the data and the distribution's general characteristics.

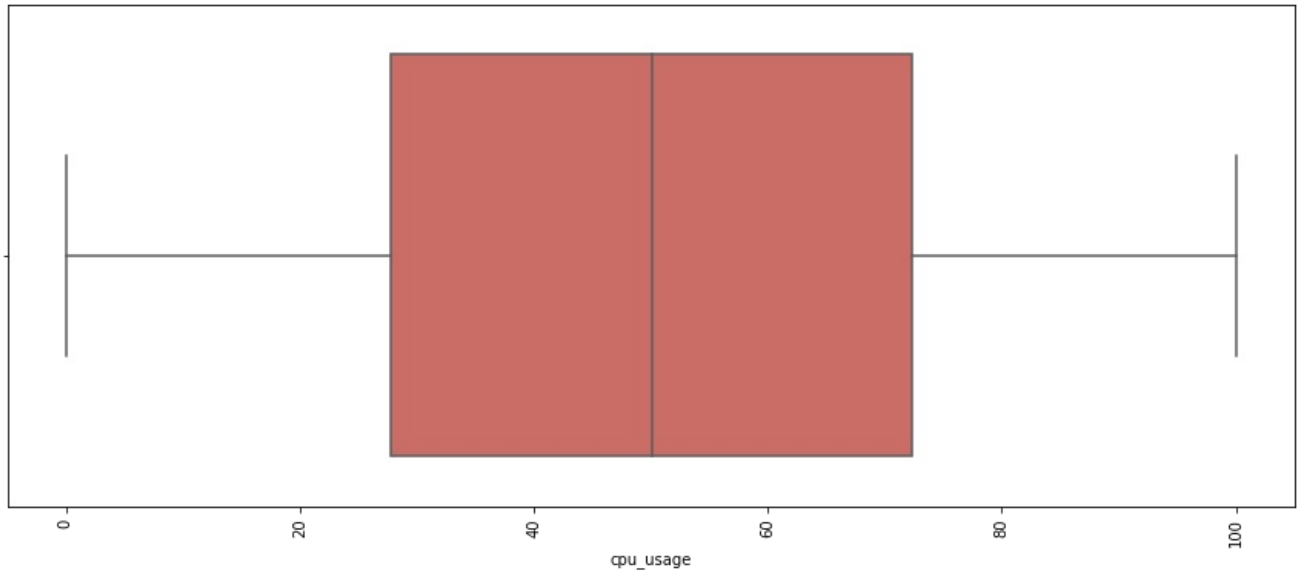
4. **Customization:** Histograms can be customized by adjusting the number of bins and other display properties to emphasize specific aspects of the data distribution.

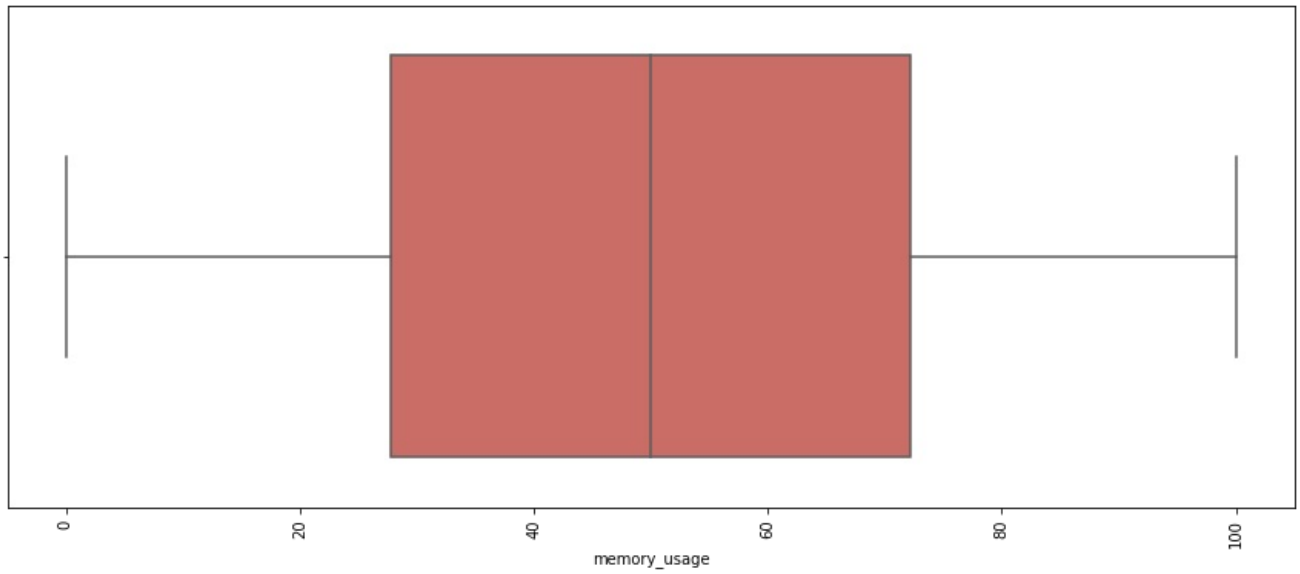
Distplot (Distribution Plot):

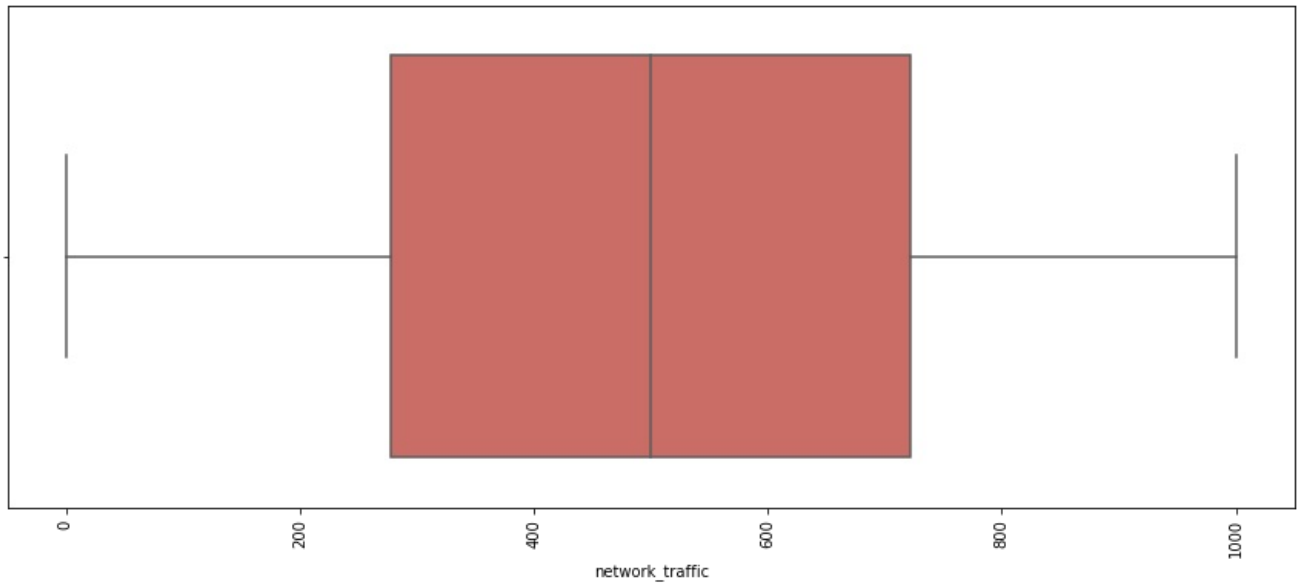
1. **Visualization Method:** A distplot, on the other hand, is a more comprehensive visualization tool provided by libraries like Seaborn, which is built on top of Matplotlib. It combines multiple elements, including a histogram, kernel density estimate (KDE), and rug plot.
2. **Components:** A distplot typically includes a histogram, which provides a bar plot of the data's distribution, a KDE plot, which is a smoothed estimate of the probability density function, and a rug plot, which places small vertical lines at each data point along the x-axis.
3. **Usage:** Distplots are used to provide a more detailed and informative representation of data distribution. They combine the information from a histogram and a KDE to show the data's distribution shape and the underlying probability density.
4. **Customization:** Distplots often offer more customization options for modifying the appearance of the histogram, KDE, and rug plot. You can adjust bandwidth, colors, and other visual elements to better suit your needs.

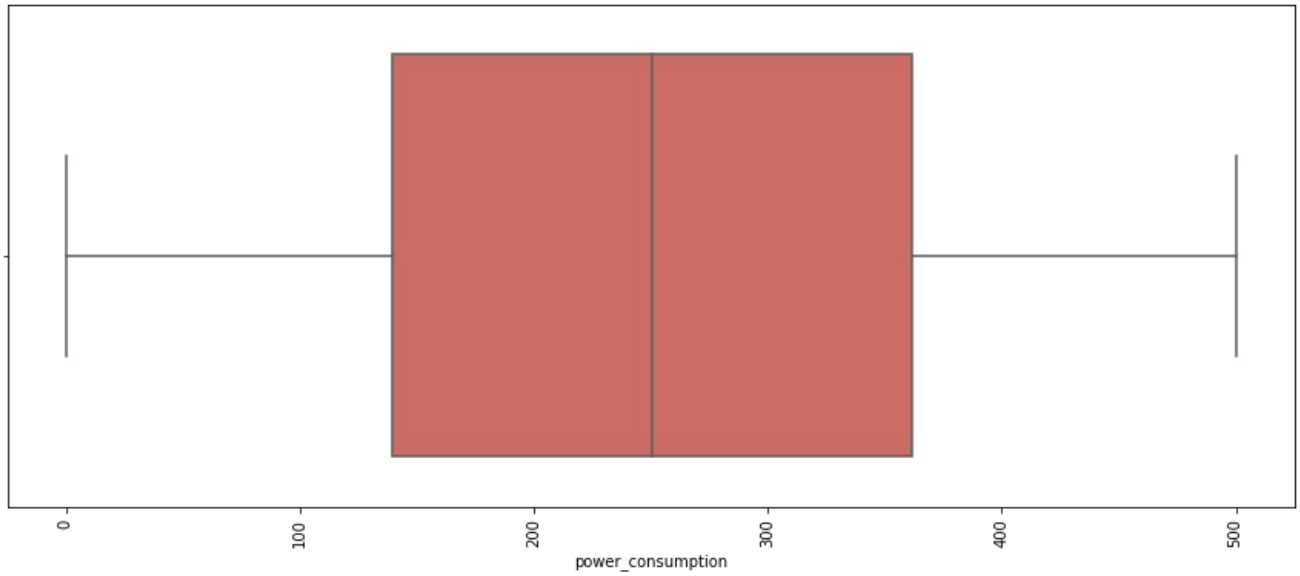
In summary, the main difference between a histogram and a distplot lies in their level of detail and the additional information they provide. A histogram is a basic tool for showing data distribution, while a distplot is a more feature-rich visualization tool that includes a histogram, KDE, and a rug plot. The choice between the two depends on your specific visualization requirements and how much detail you want to convey in your data analysis.

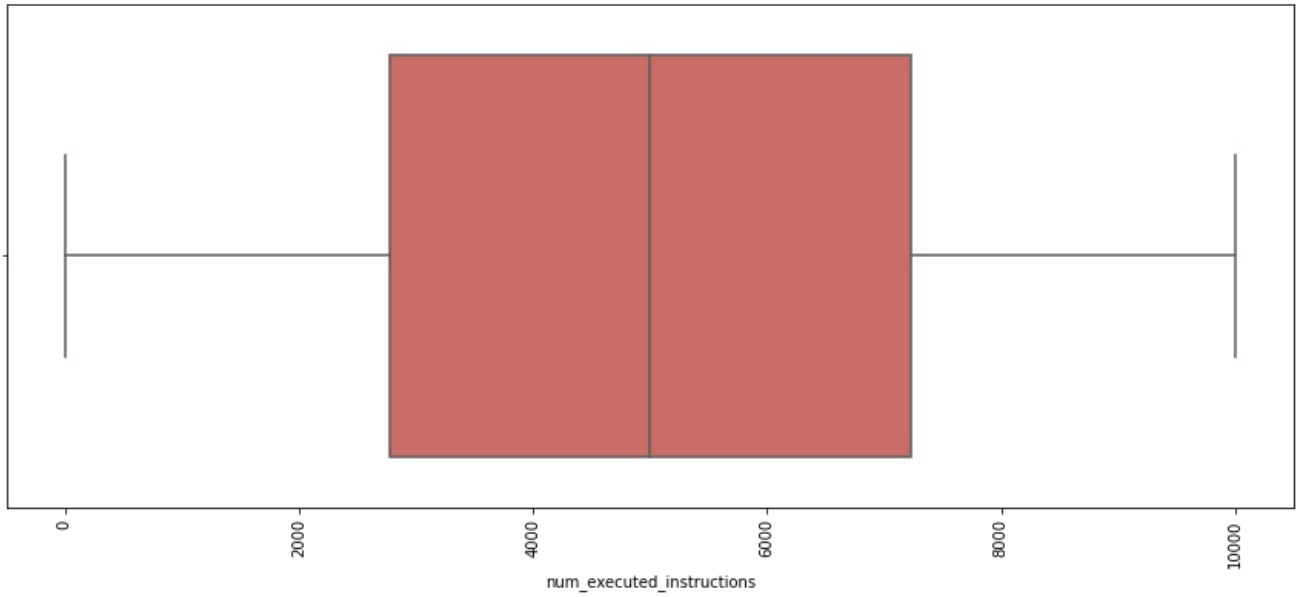
In [39]:
```
for i in numerical_columns:
    plt.figure(figsize=(15,6))
    sns.boxplot(x = df[i], data = df, palette = 'hls')
    plt.xticks(rotation = 90)
    plt.show()
```

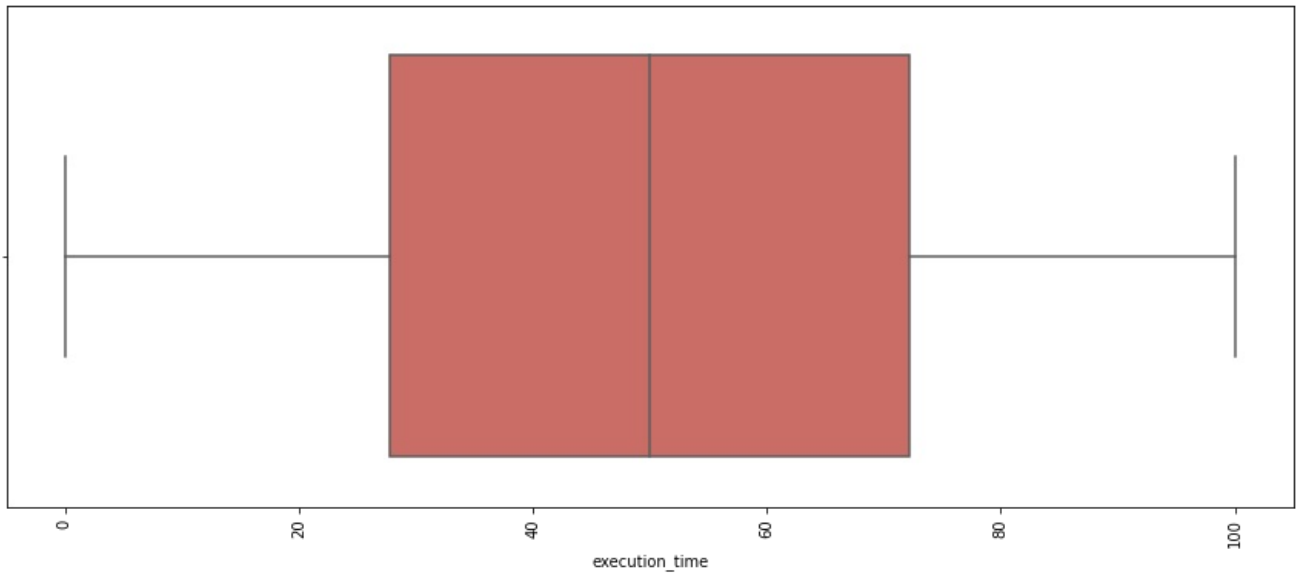


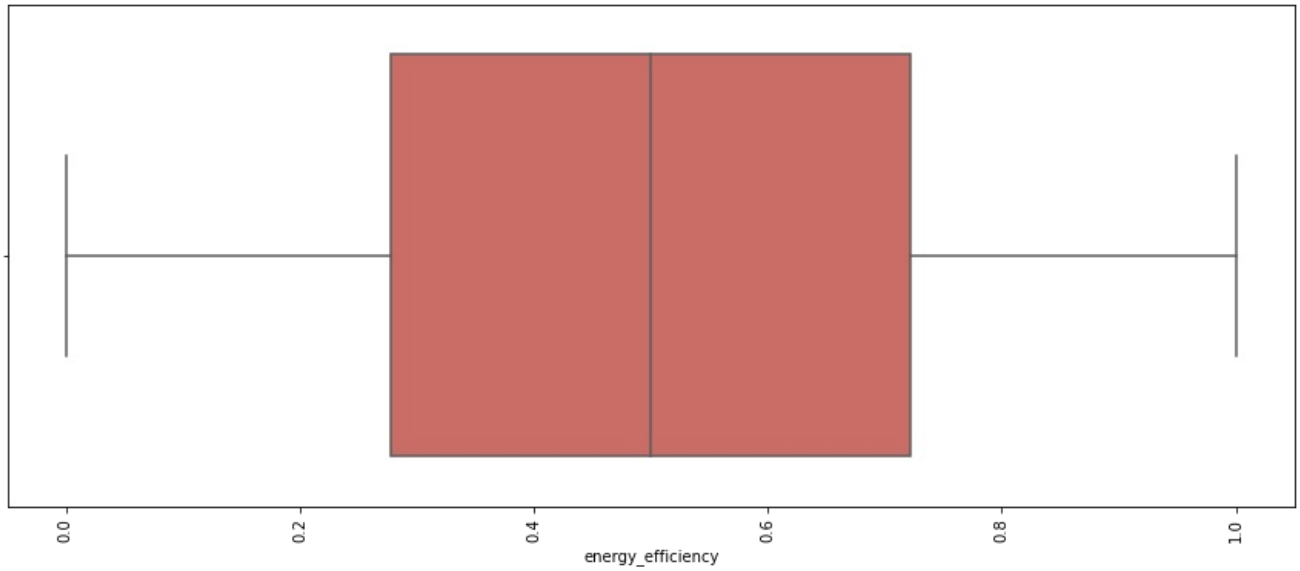


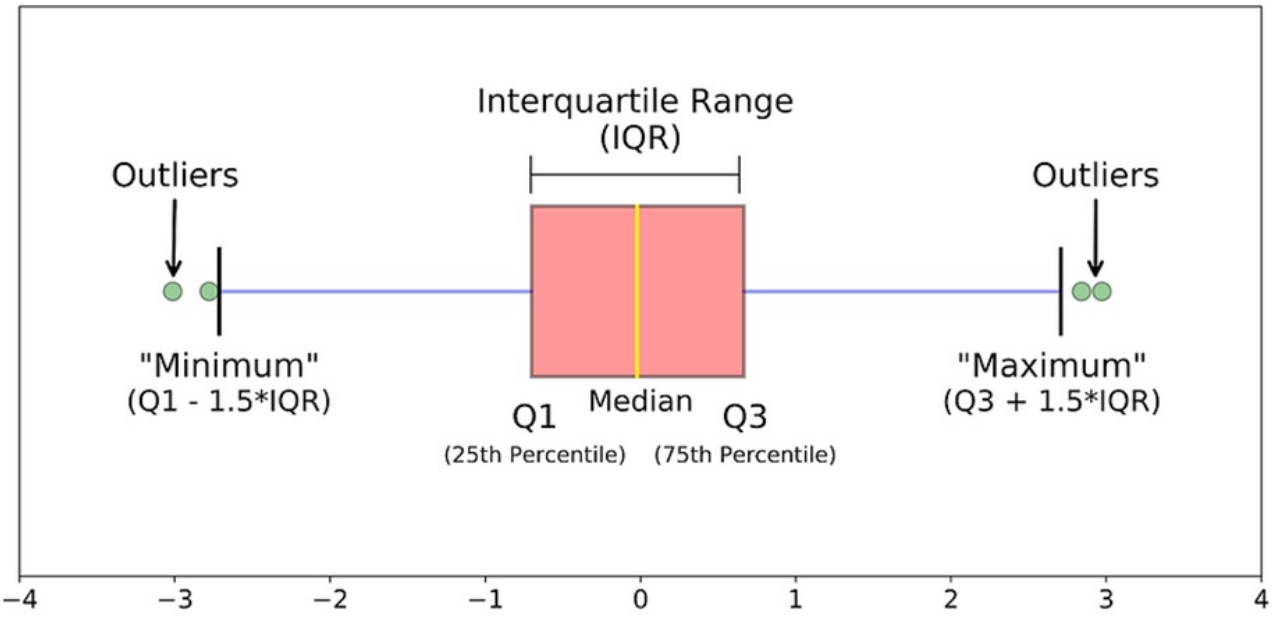

0
200
400
600
800
1000
network_traffic


0
100
200
300
400
500
power_consumption


0
2000
4000
6000
8000
10000
num_executed_instructions

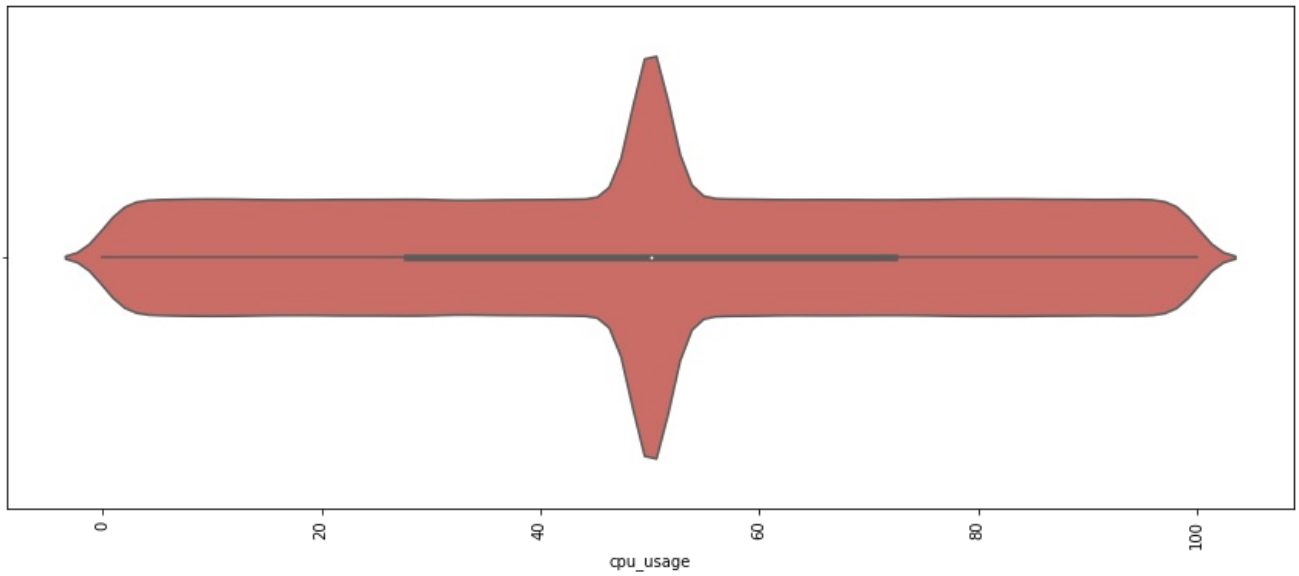


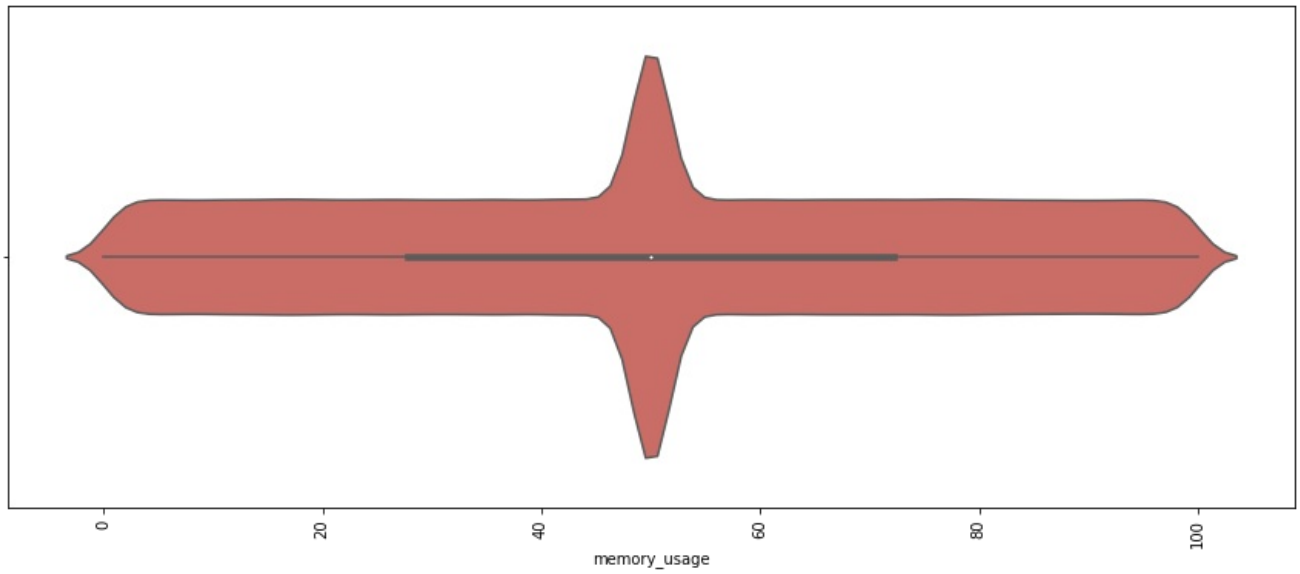


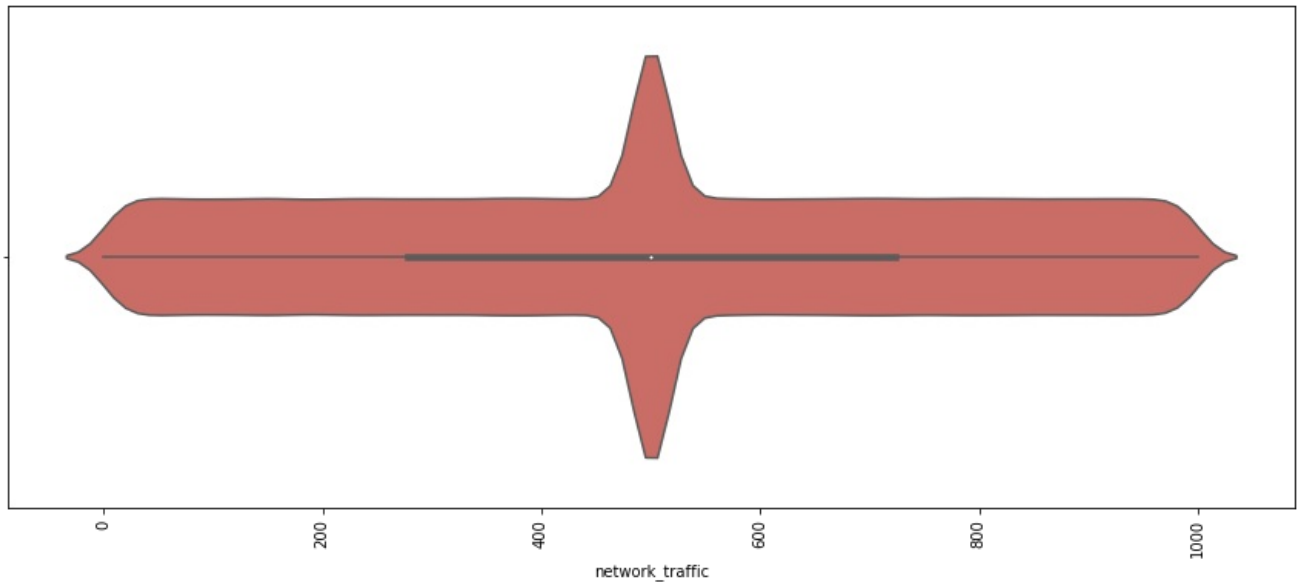


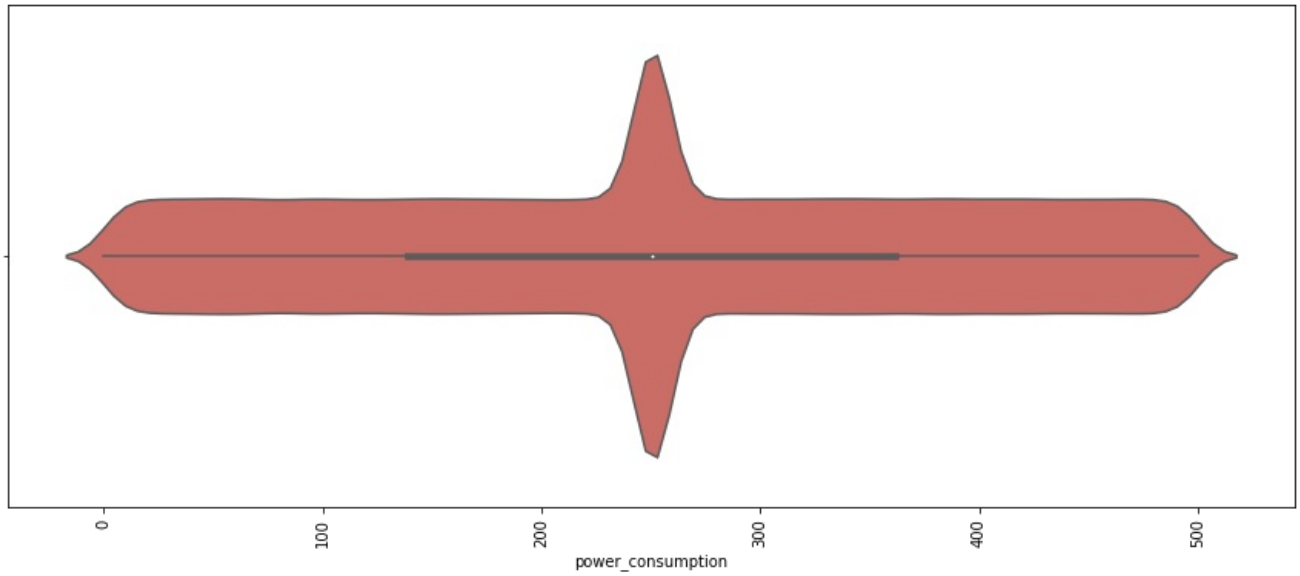
In [40]:
```
for i in numerical_columns:
```

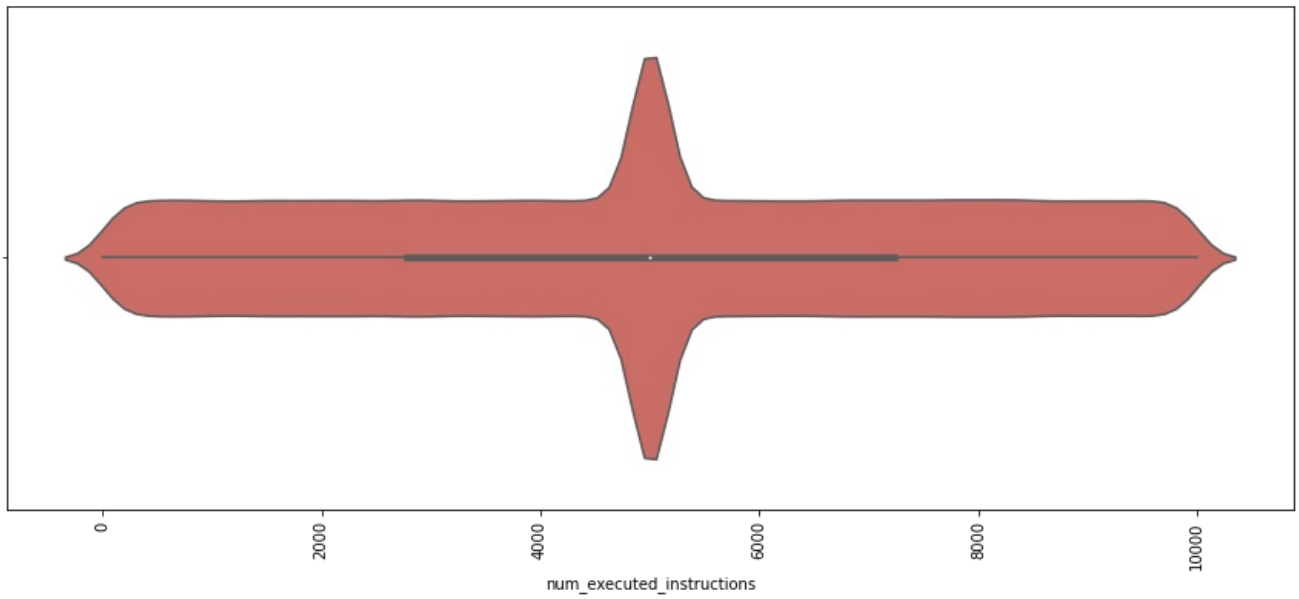
```
    plt.figure(figsize=(15,6))
    sns.violinplot(x = df[i], data = df, palette = 'hls')
    plt.xticks(rotation = 90)
    plt.show()
```

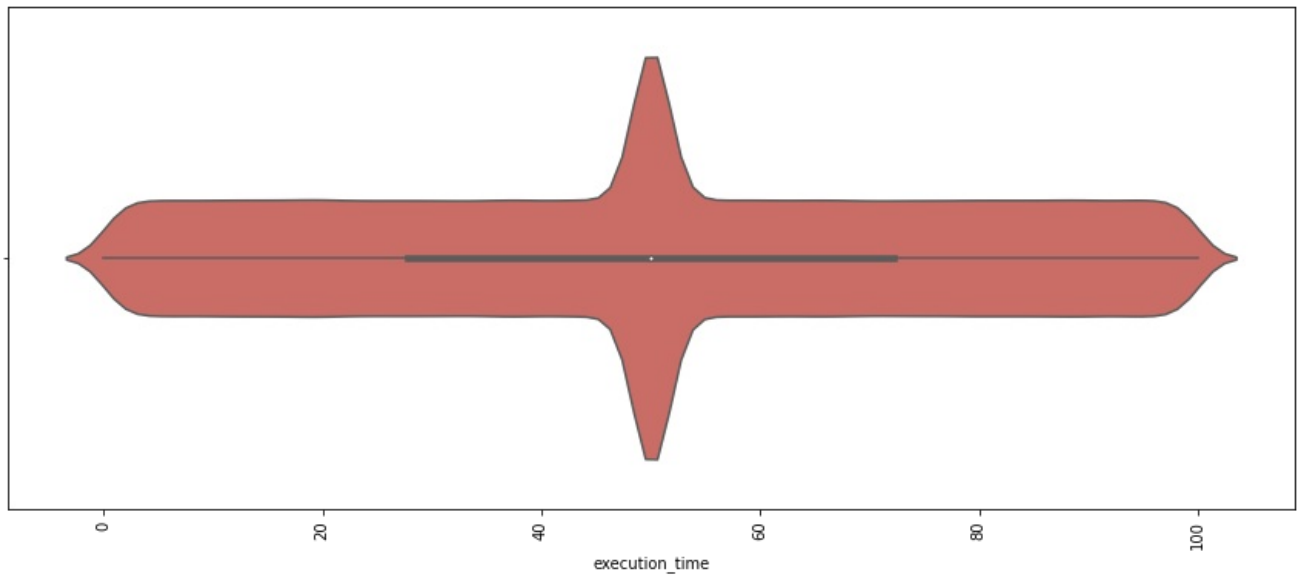


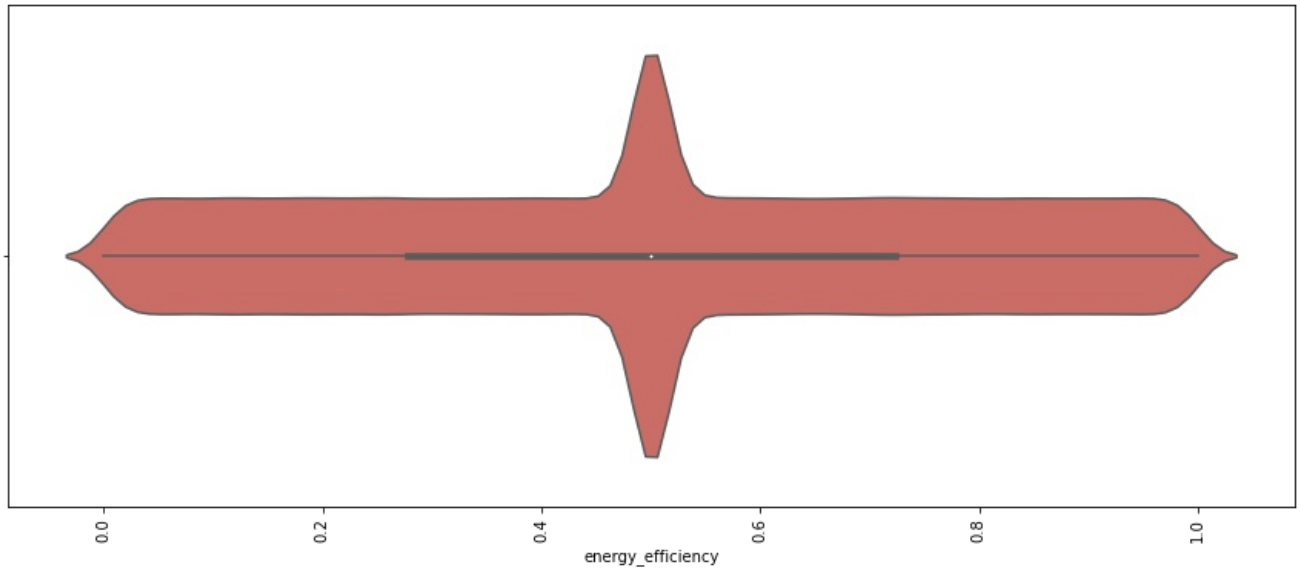





0
100
200
300
400
500
power_consumption


0
2000
4000
6000
8000
10000
num_executed_instructions


0
20
40
60
80
100
execution_time



Violin plots and box plots are both used for visualizing the distribution of data and summarizing key statistical parameters, but they have different characteristics and purposes. Here are the main differences between violin plots and box plots:

Box Plot (Box-and-Whisker Plot):

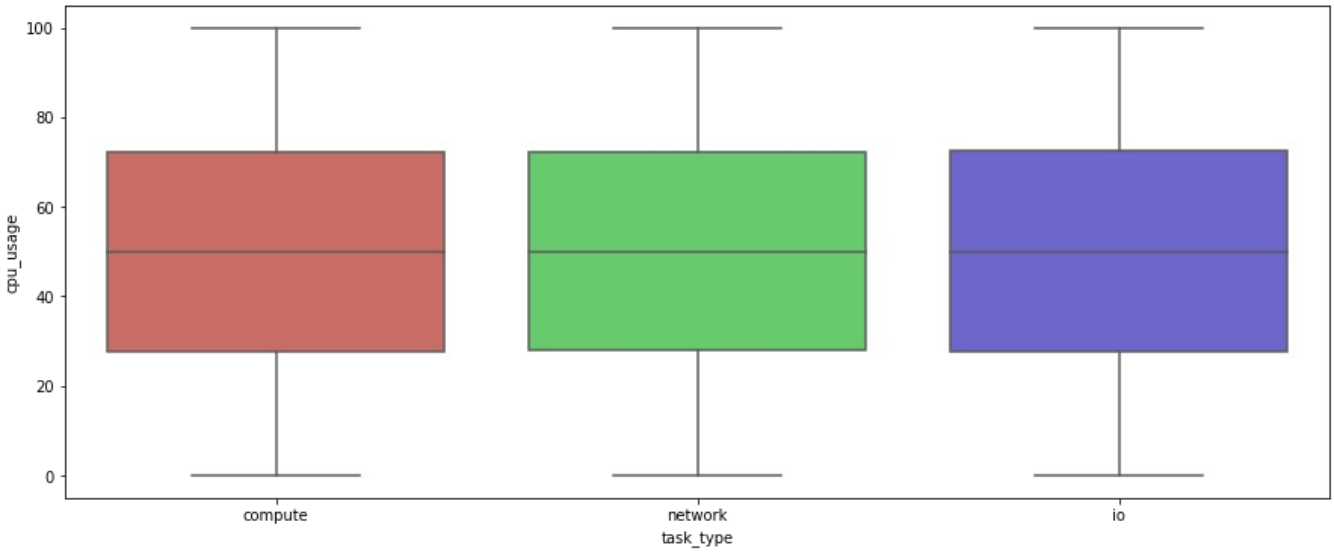
1. **Shape:** A box plot displays the distribution of data in a compact, rectangular box. It typically includes a box (hence the name) with a vertical line (the median) inside it and "whiskers" extending from the box.
2. **Key Information:** A box plot provides information about the median, quartiles (25th and 75th percentiles), and potential outliers in the data. The whiskers indicate the data's spread, and any data points outside the whiskers are plotted as individual points, representing potential outliers.
3. **Representation:** It is primarily used to display summary statistics and to identify the presence of outliers in the data. It is particularly useful for comparing the distribution of data in multiple groups or categories.
4. **Simplicity:** Box plots are simple and straightforward, making them easy to interpret. They do not provide detailed information about the shape of the data distribution.

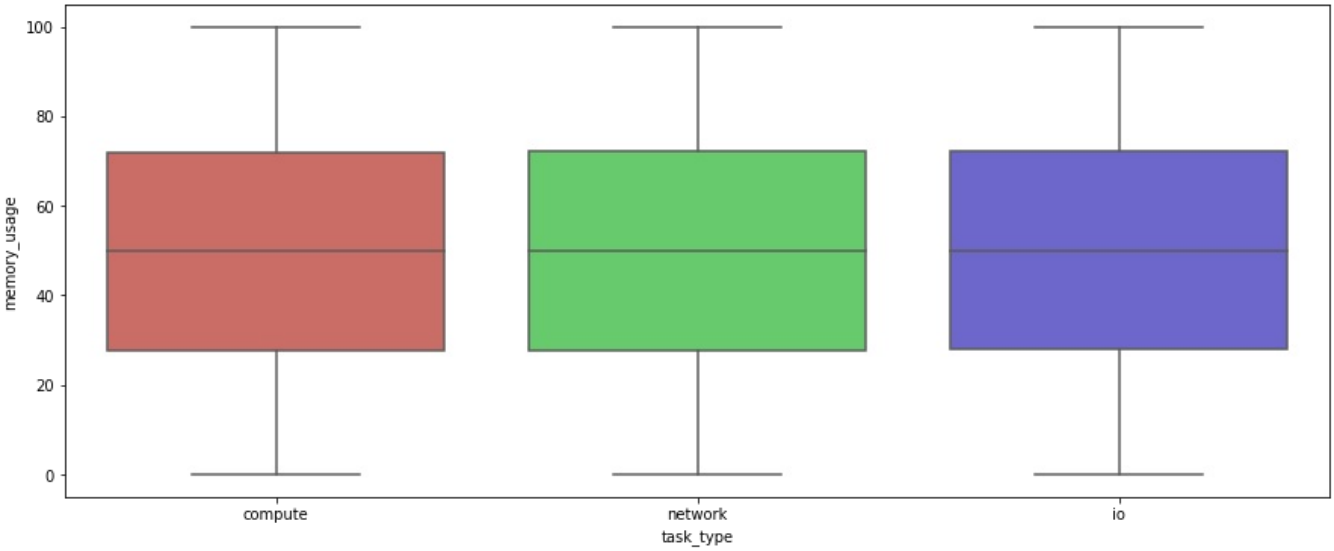
Violin Plot:

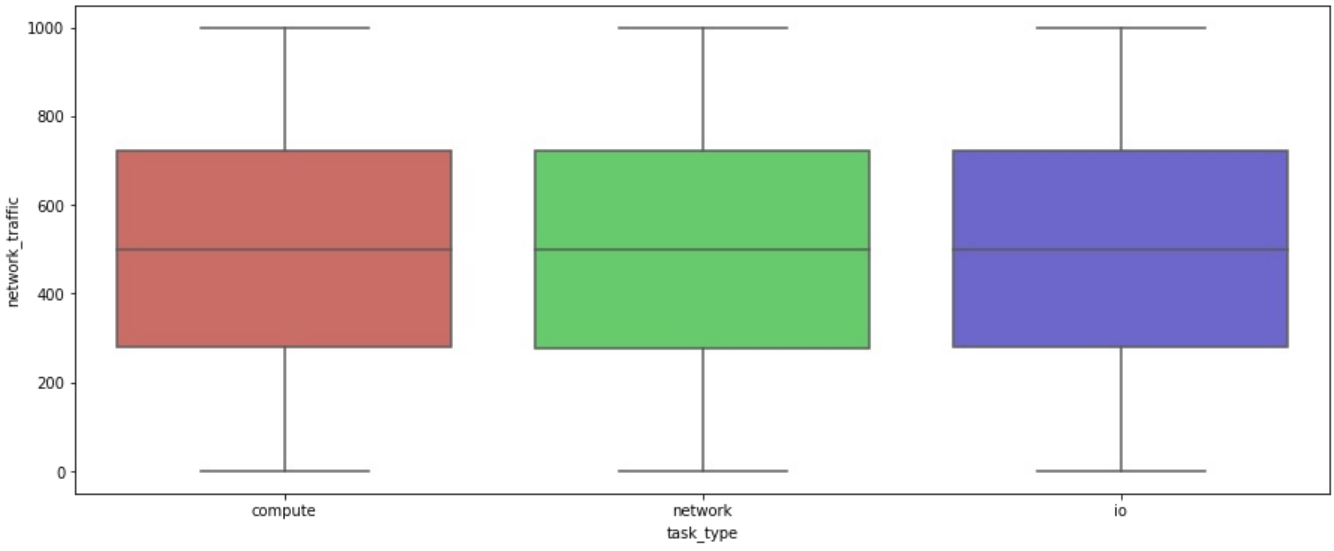
1. **Shape:** A violin plot is a combination of a box plot and a kernel density plot. It consists of a central box (similar to a box plot) and a pair of symmetrical density plots on each side. These density plots resemble violins, which is where the name comes from.
2. **Key Information:** A violin plot provides information about the median, quartiles, and the data's overall distribution. The width of the violin at a specific point represents the density of data at that location. This provides more information about the shape of the data distribution compared to a box plot.
3. **Representation:** Violin plots are useful when you want to visualize the entire data distribution, including its shape and any multimodality (multiple peaks). They are effective for comparing distributions between different groups.
4. **Detail:** Violin plots offer a more detailed view of the data distribution than box plots. They are particularly useful for identifying and illustrating the presence of multiple modes or complex shapes in the data.

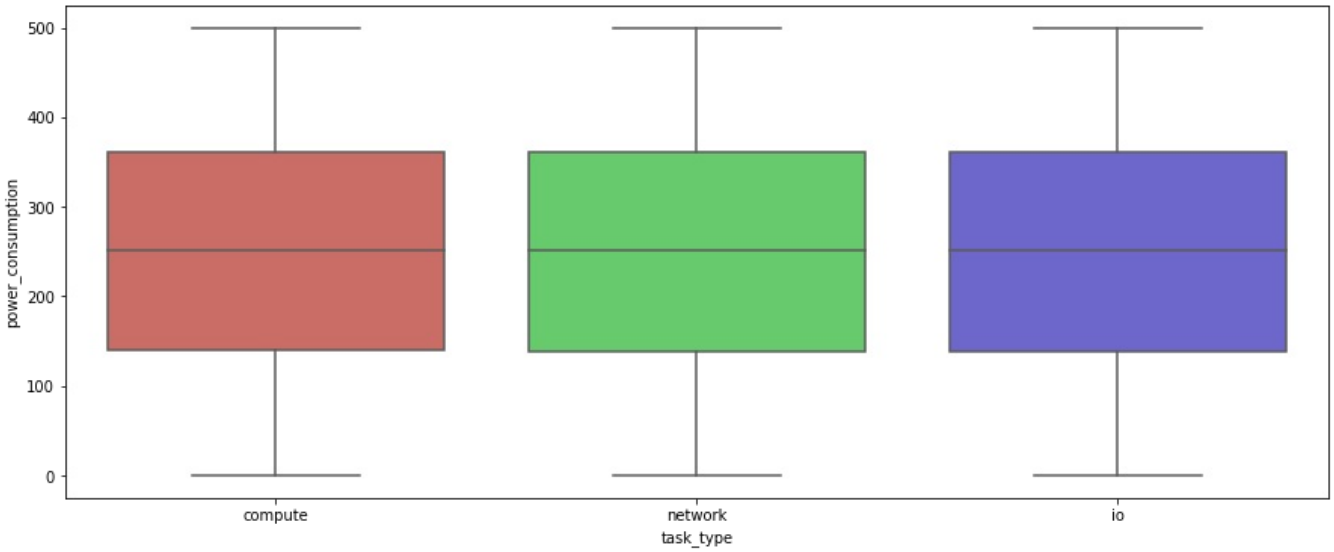
In summary, the choice between a violin plot and a box plot depends on your specific data visualization goals. Use a box plot when you want a simple summary of key statistics and to identify potential outliers. Use a violin plot when you want a more detailed view of the data distribution, especially if it is multimodal or has complex shapes. Violin plots are often preferred when exploring the nuances of data, while box plots are used for a quick overview of central tendencies and variability.

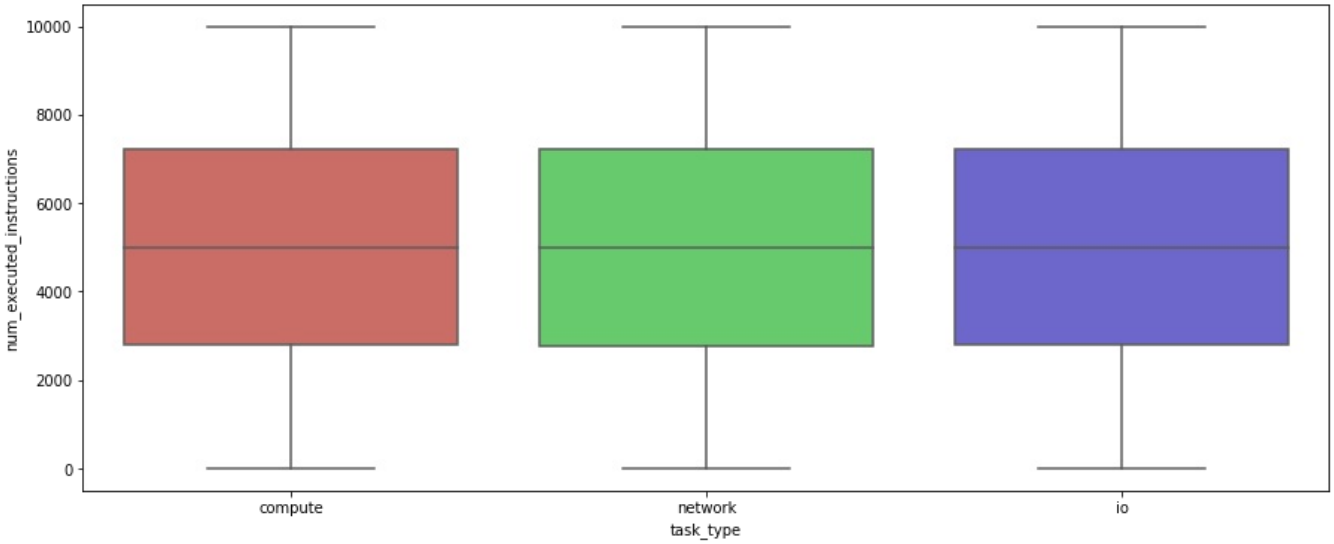
In [41]:
```
for i in categorical_columns:
    for j in numerical_columns:
        plt.figure(figsize=(15,6))
        sns.boxplot(x = df[i], y = df[j], data = df, palette = 'hls')
        plt.show()
```

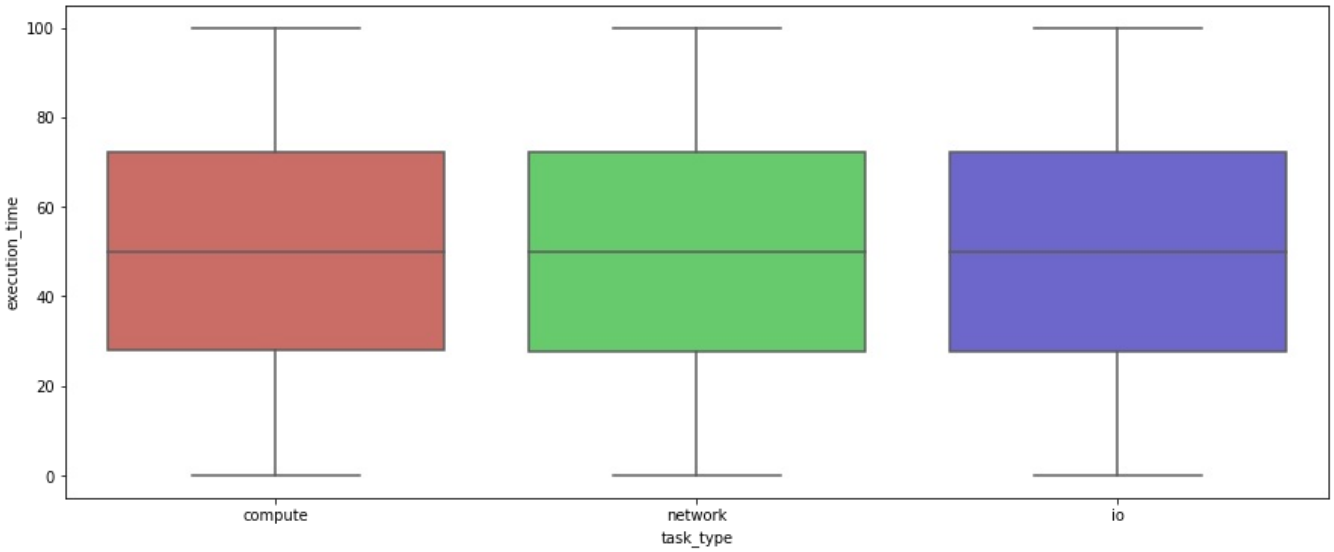

cpu_usage
100
80
60
40
20
0
compute
network
io
task_type

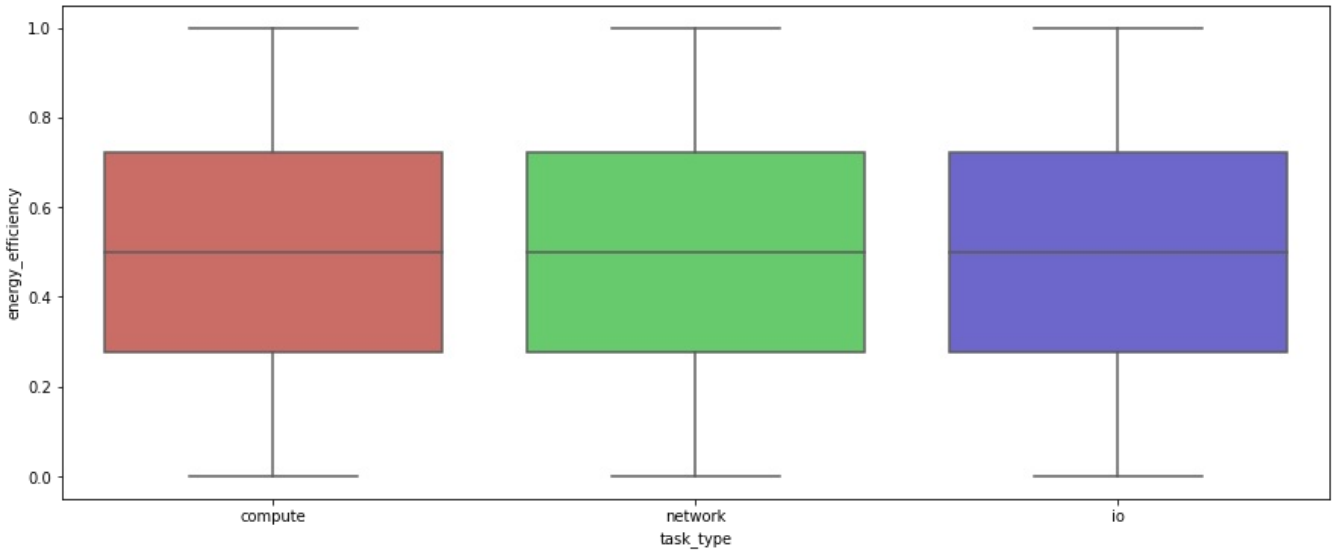

memory_usage
100
80
60
40
20
0
compute
network
io
task_type

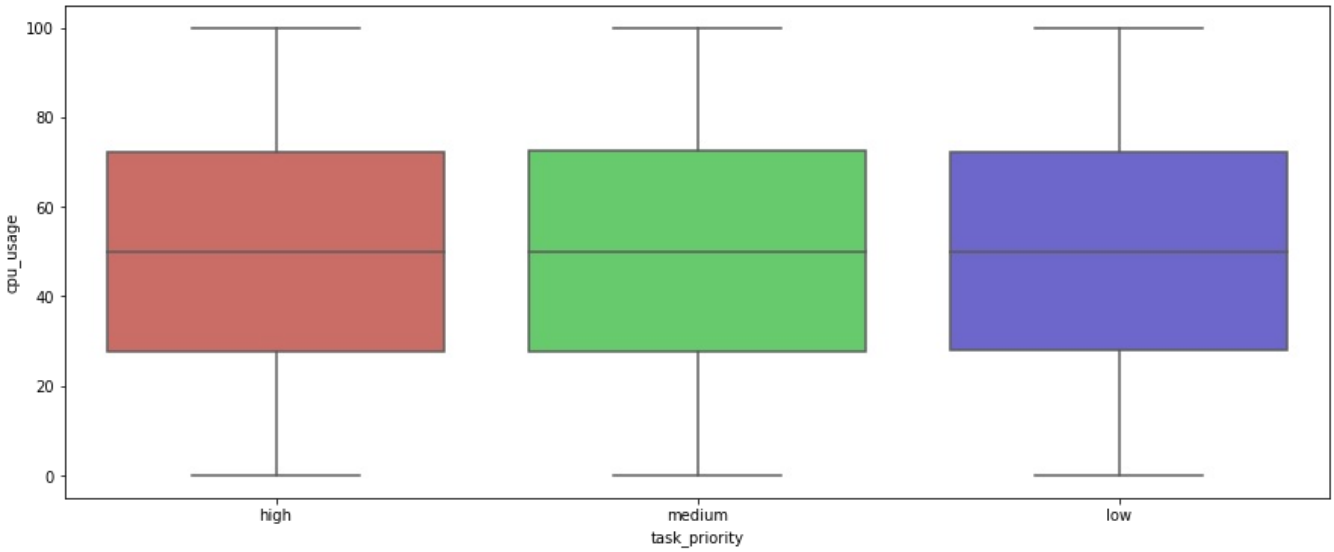

network_traffic
1000
800
600
400
200
0
compute
network
io
task_type

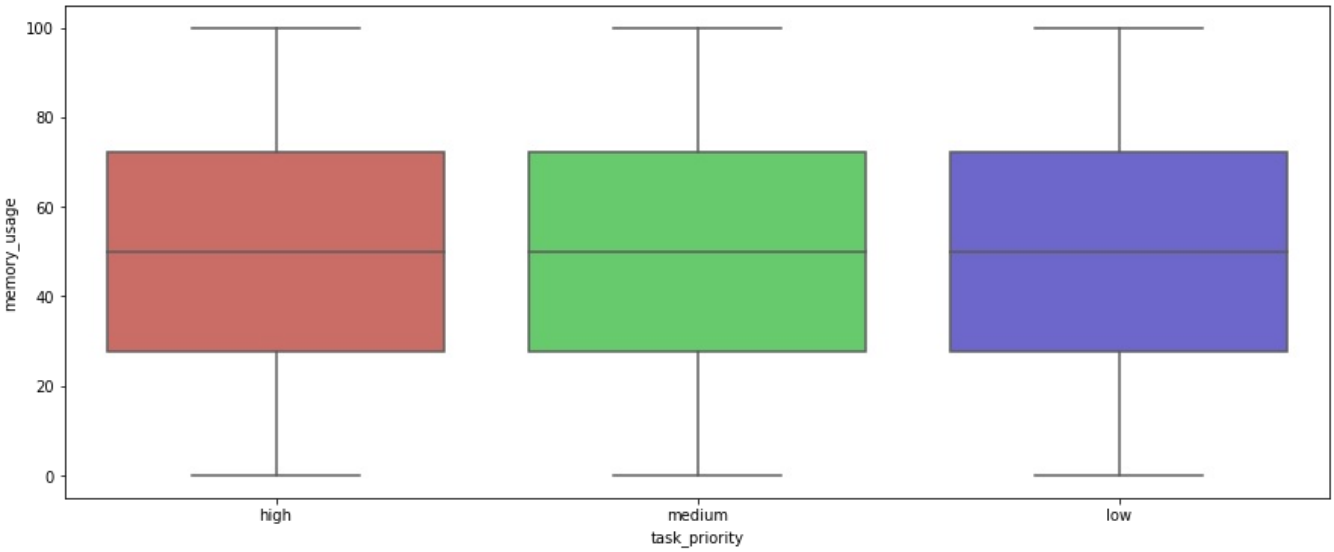

power_consumption
500
400
300
200
100
0
compute
network
io
task_type

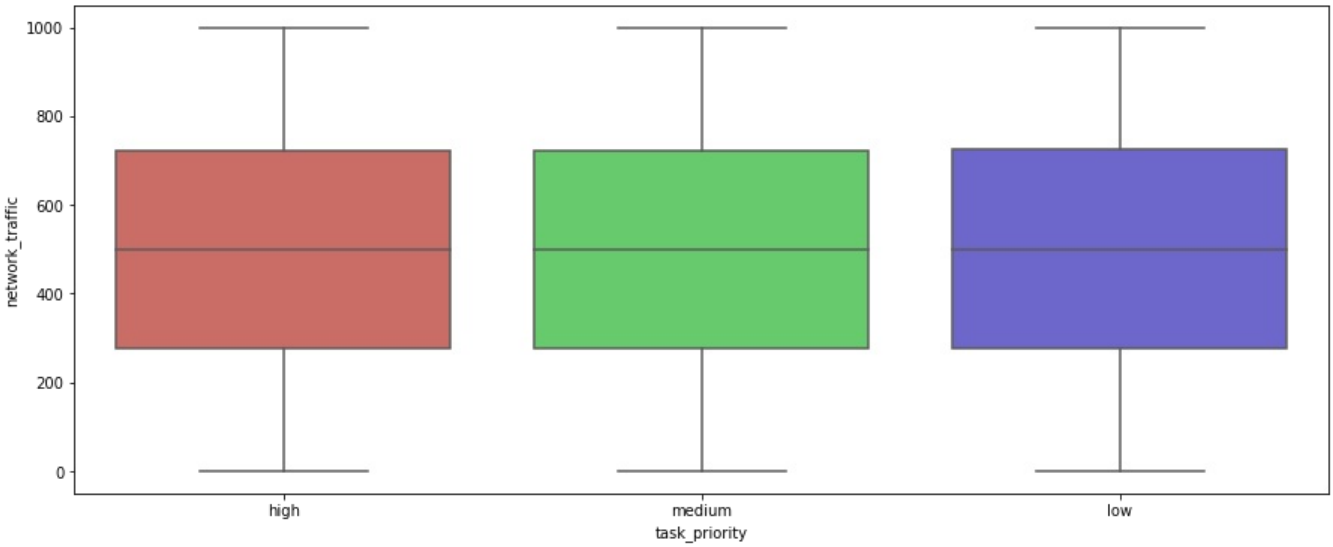

num_executed_instructions
10000
8000
6000
4000
2000
0
compute
network
io
task_type

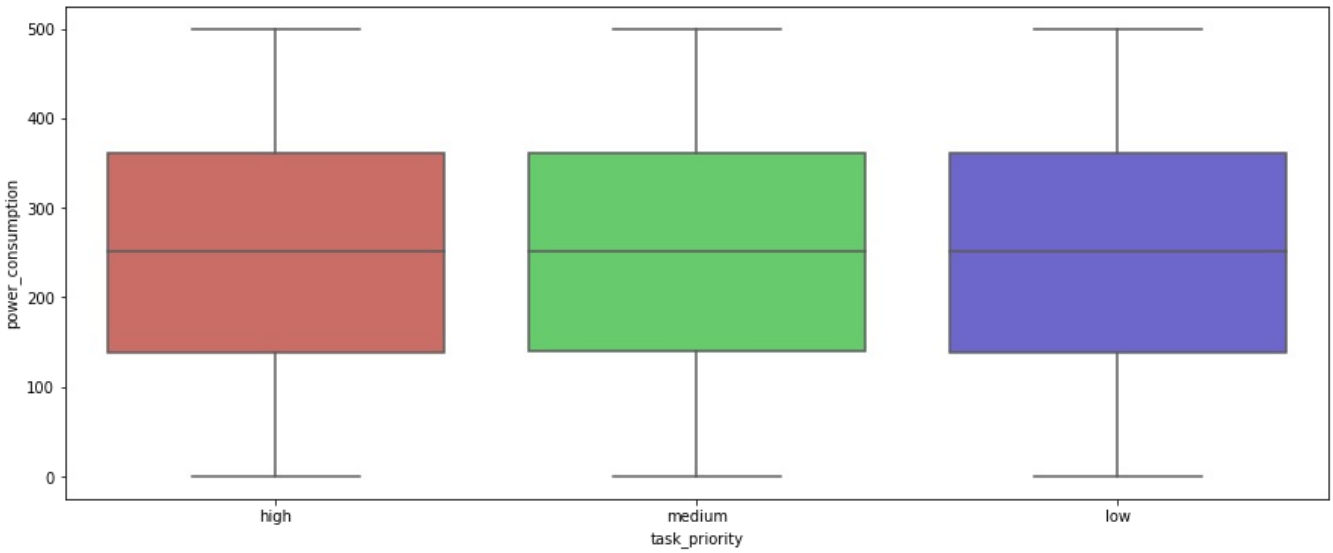

execution_time
100
80
60
40
20
0
compute
network
io
task_type

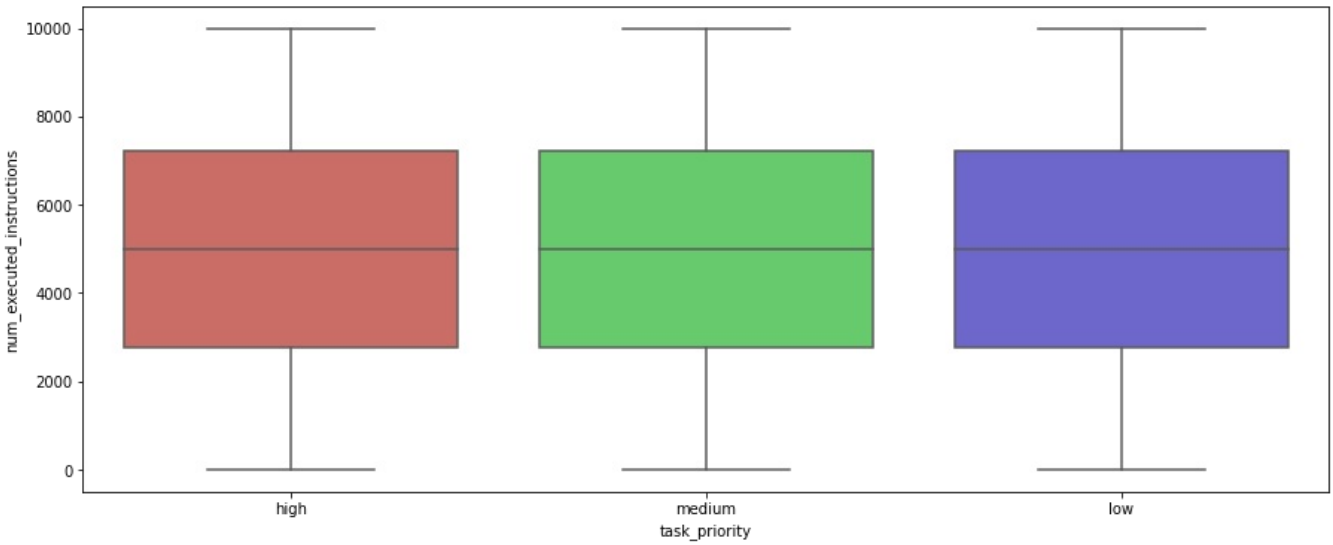

energy_efficiency
1.0
0.8
0.6
0.4
0.2
0.0
compute
network
io
task_type

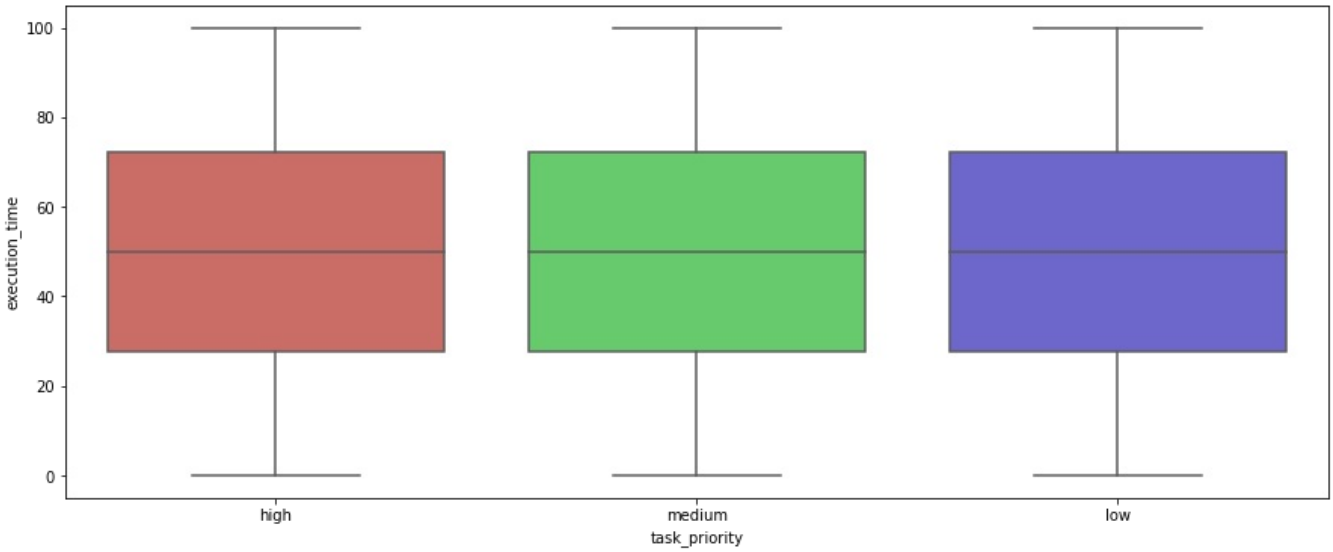

cpu_usage
100
80
60
40
20
0
high
medium
low
task_priority

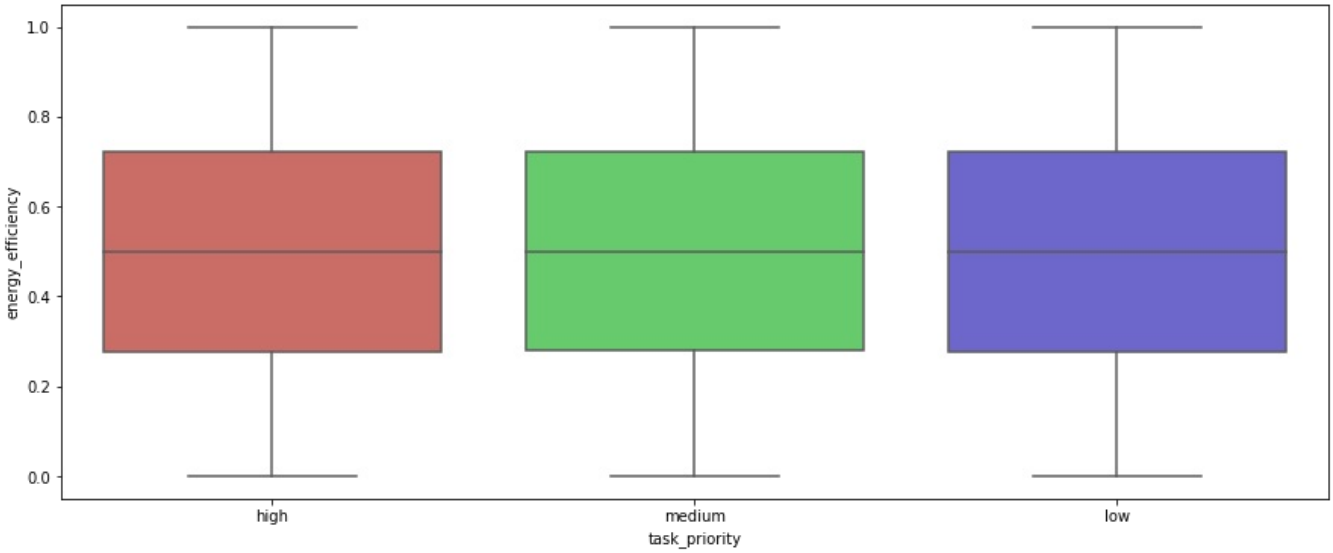

memory_usage
100
80
60
40
20
0
high
medium
low
task_priority

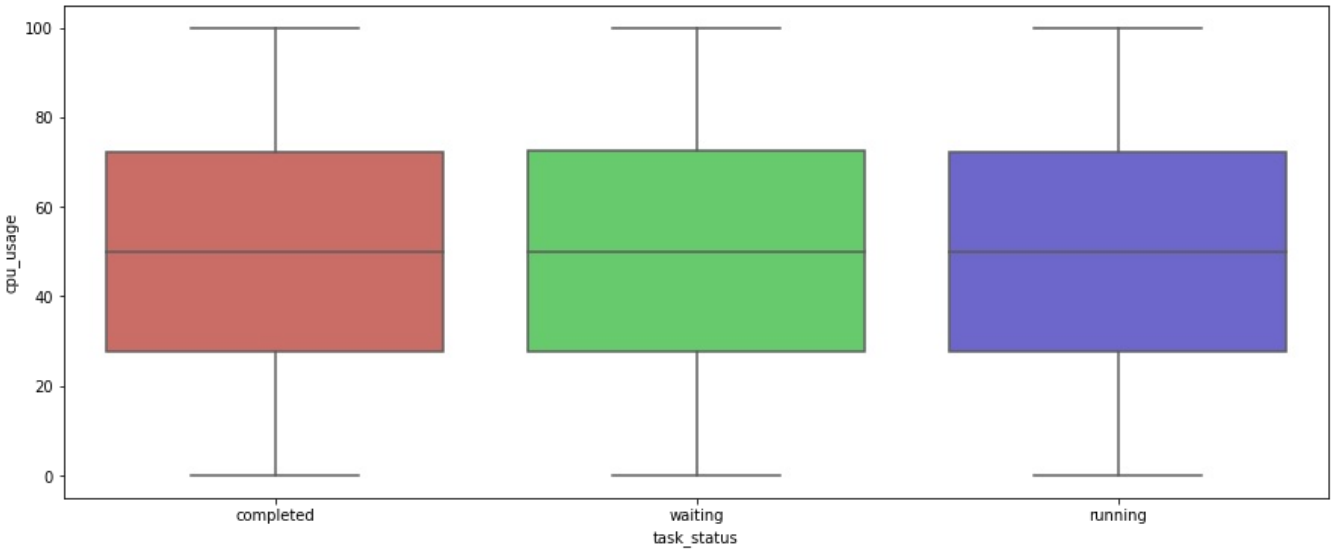

1000
800
600
400
200
0
network_traffic
high
medium
low
task_priority

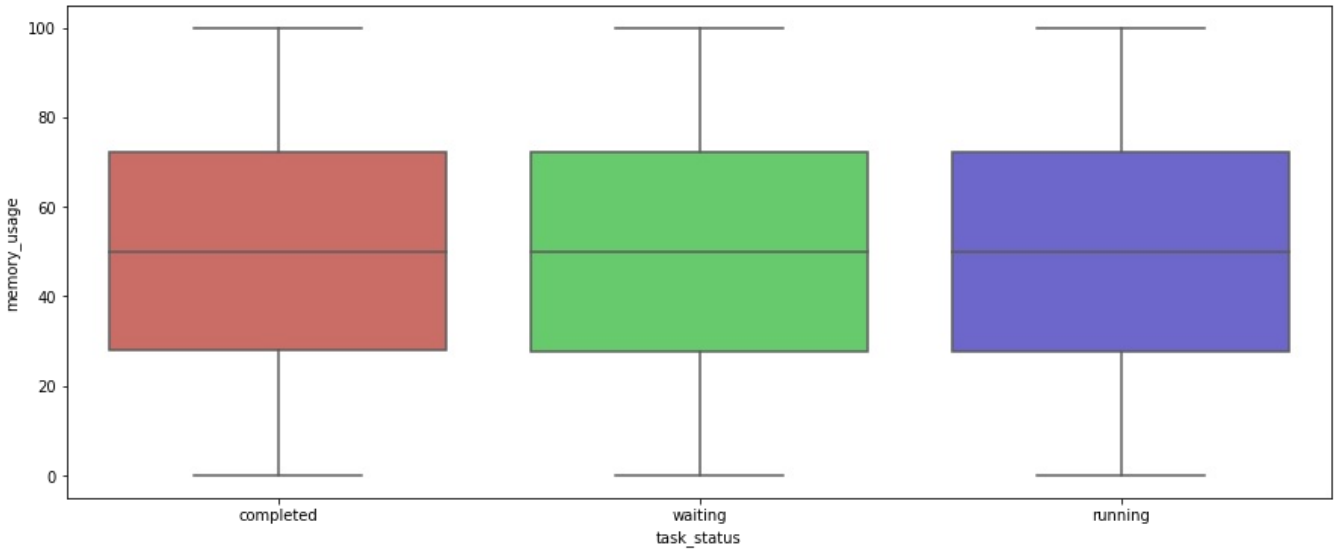

500
400
300
200
100
0
power_consumption
high
medium
low
task_priority

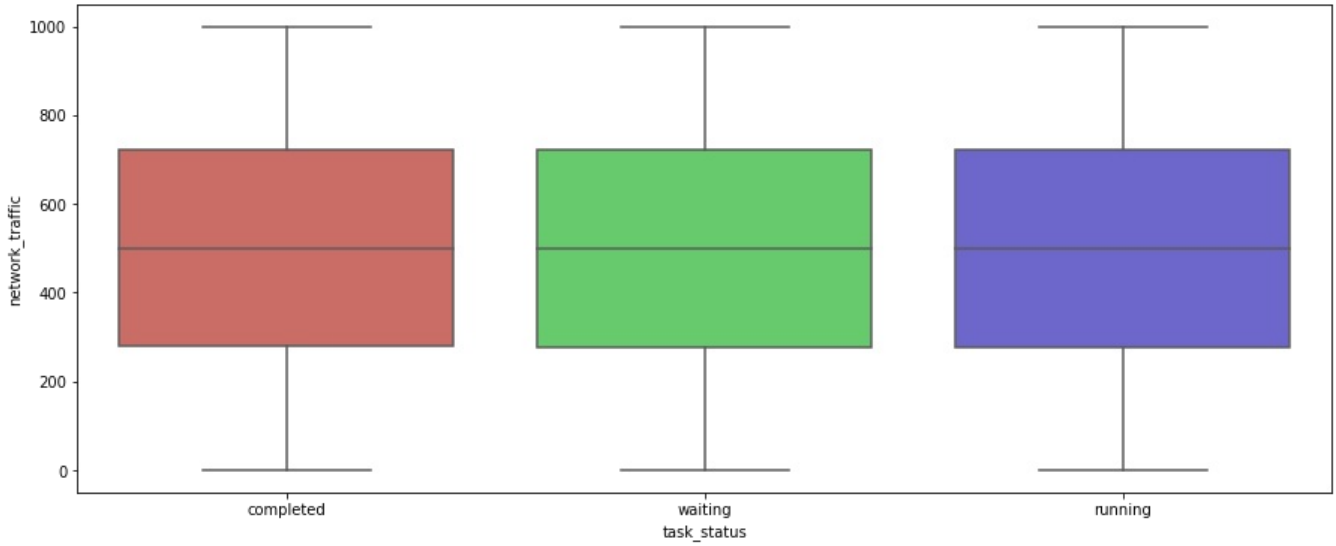

10000
8000
6000
4000
2000
0
num_executed_instructions
high
medium
low
task_priority

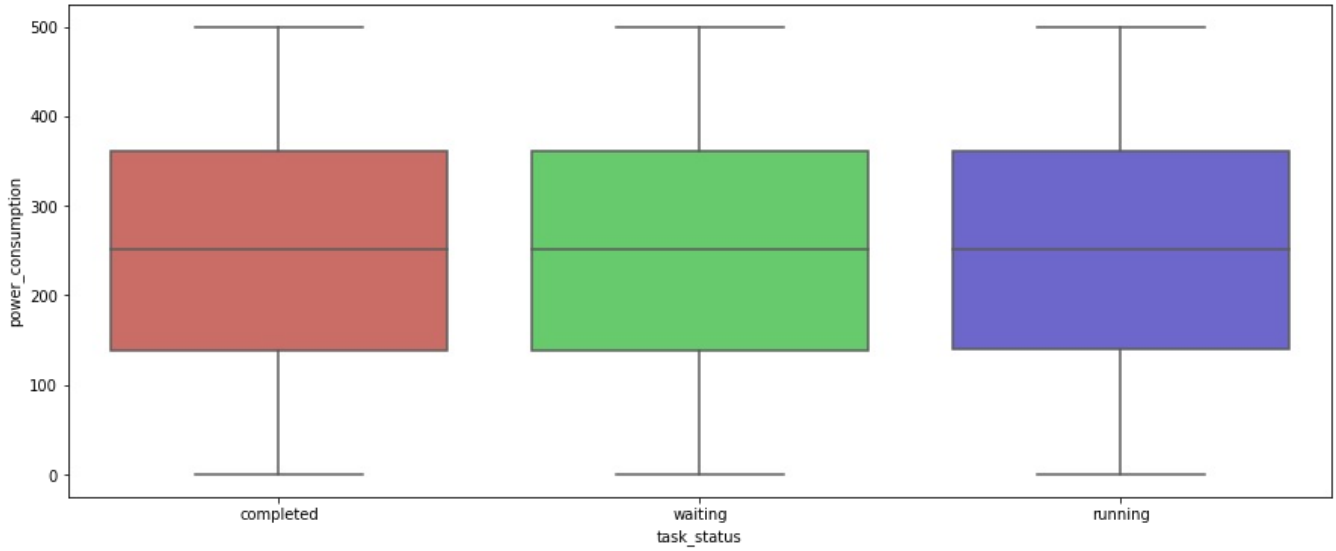

execution_time
100
80
60
40
20
0
high
medium
low
task_priority

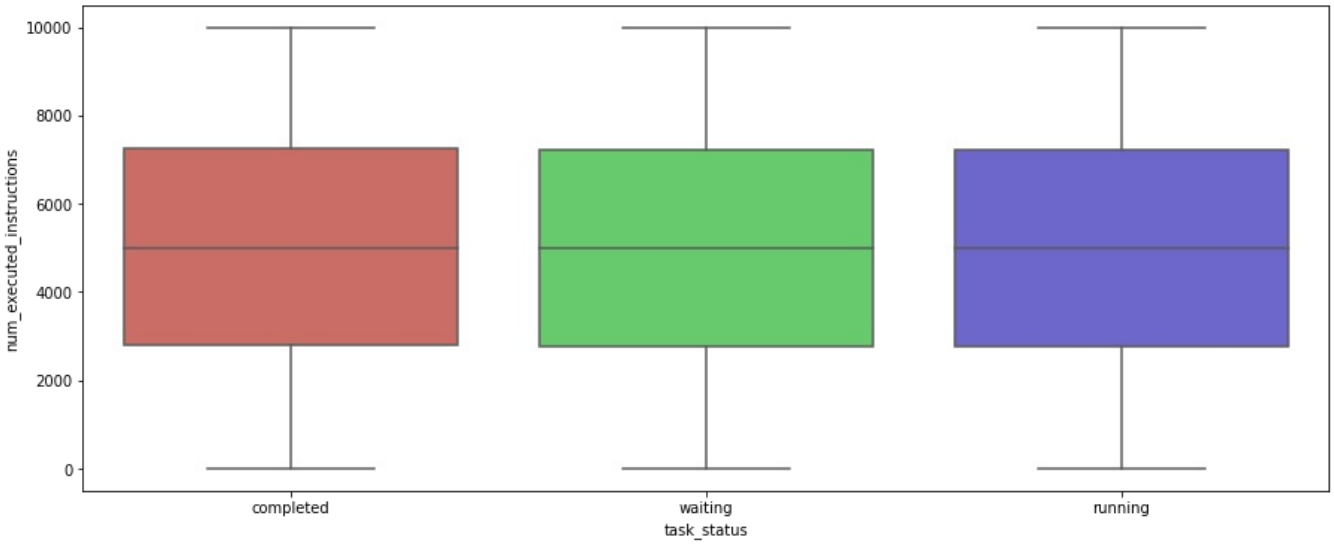

energy_efficiency
1.0
0.8
0.6
0.4
0.2
0.0
high
medium
low
task_priority

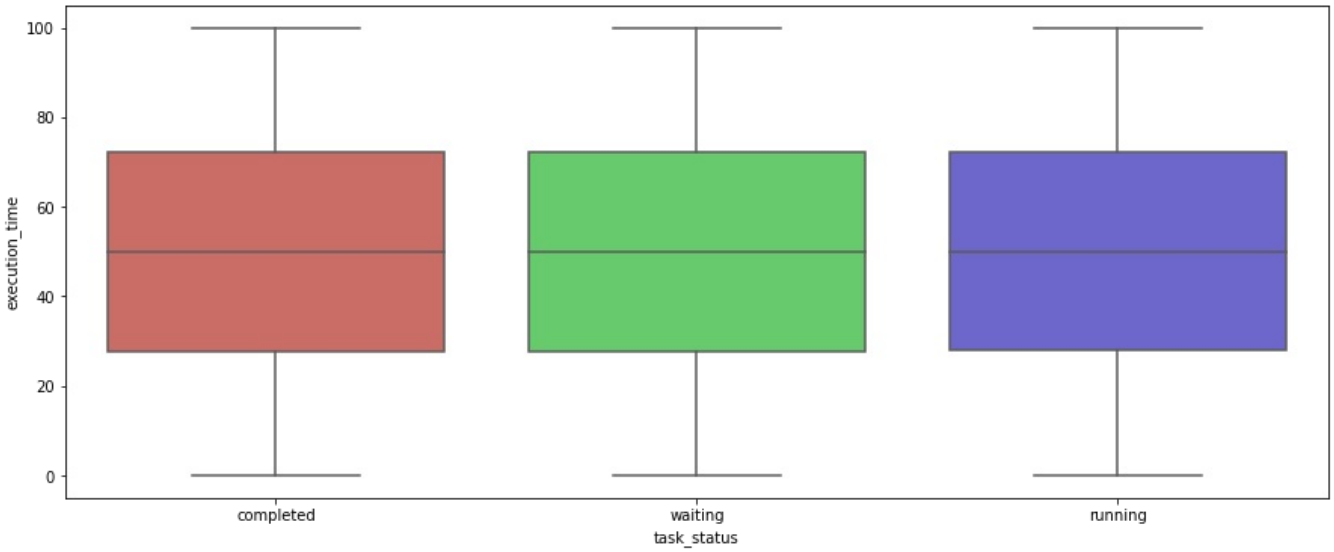

cpu_usage
100
80
60
40
20
0
completed
waiting
running
task_status

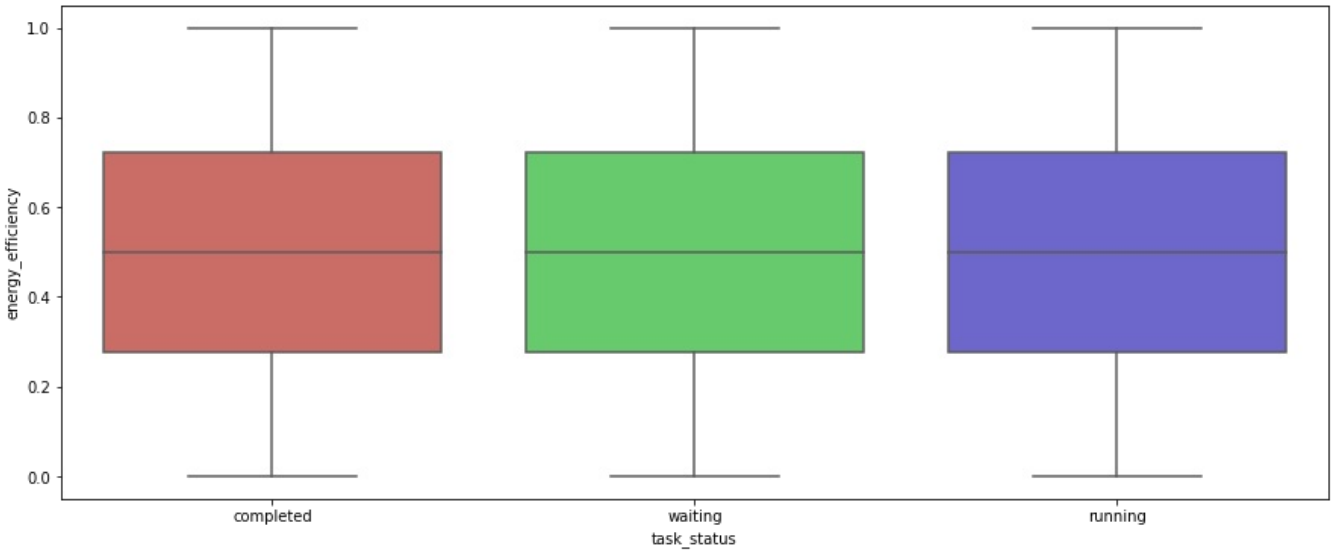

memory_usage
100
80
60
40
20
0
completed
waiting
running
task_status


network_traffic
1000
800
600
400
200
0
completed
waiting
running
task_status

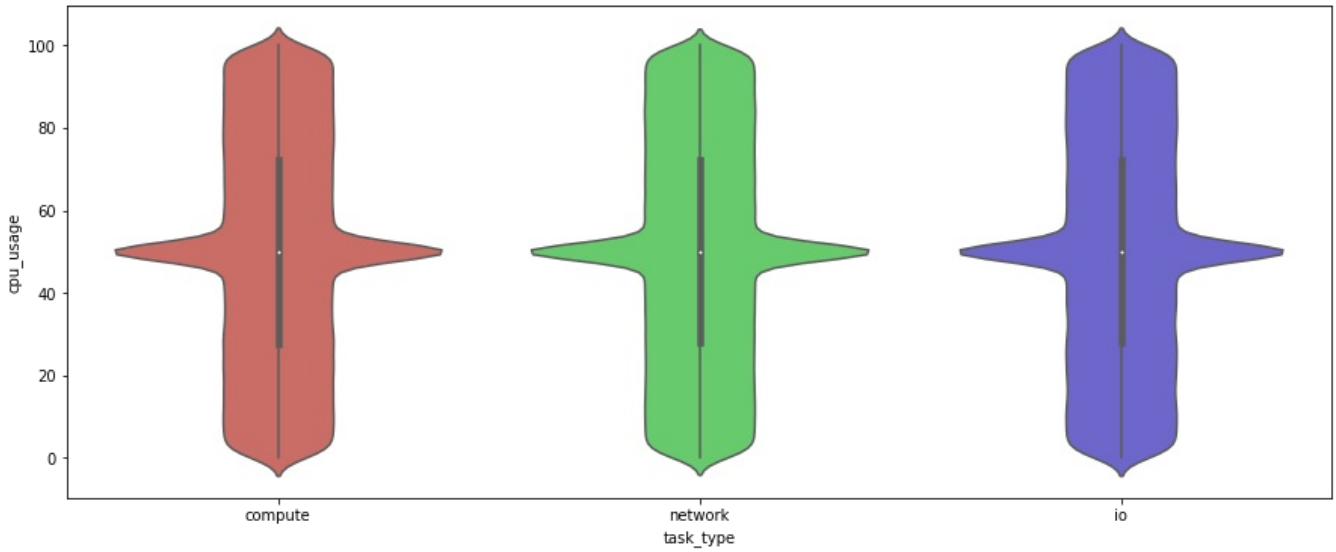

power_consumption
500
400
300
200
100
0
completed
waiting
running
task_status

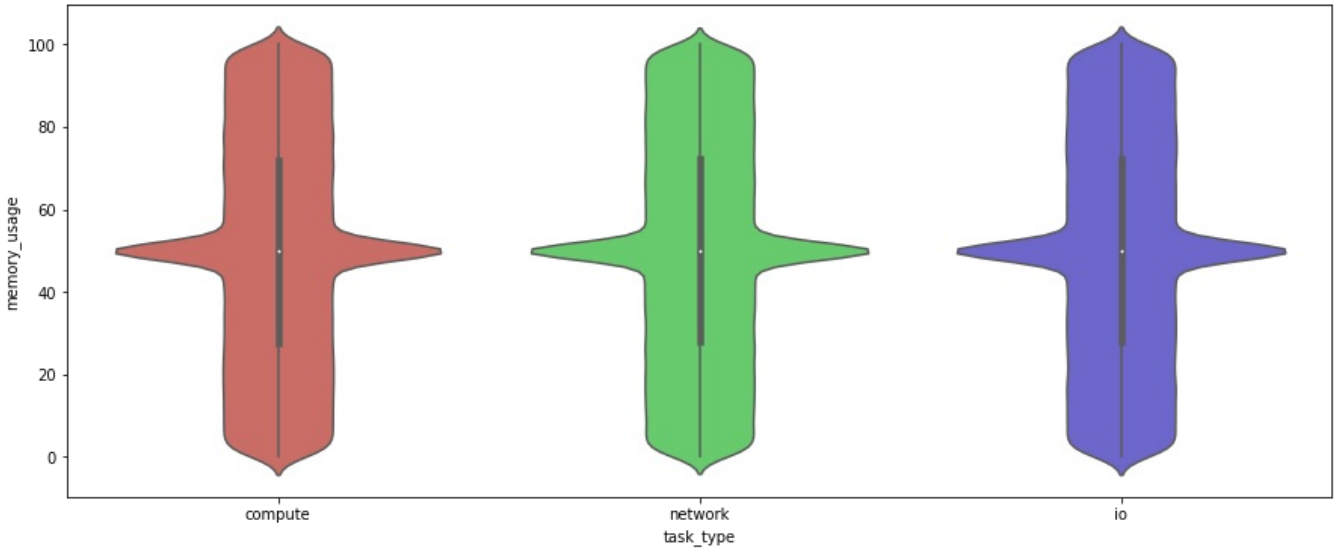


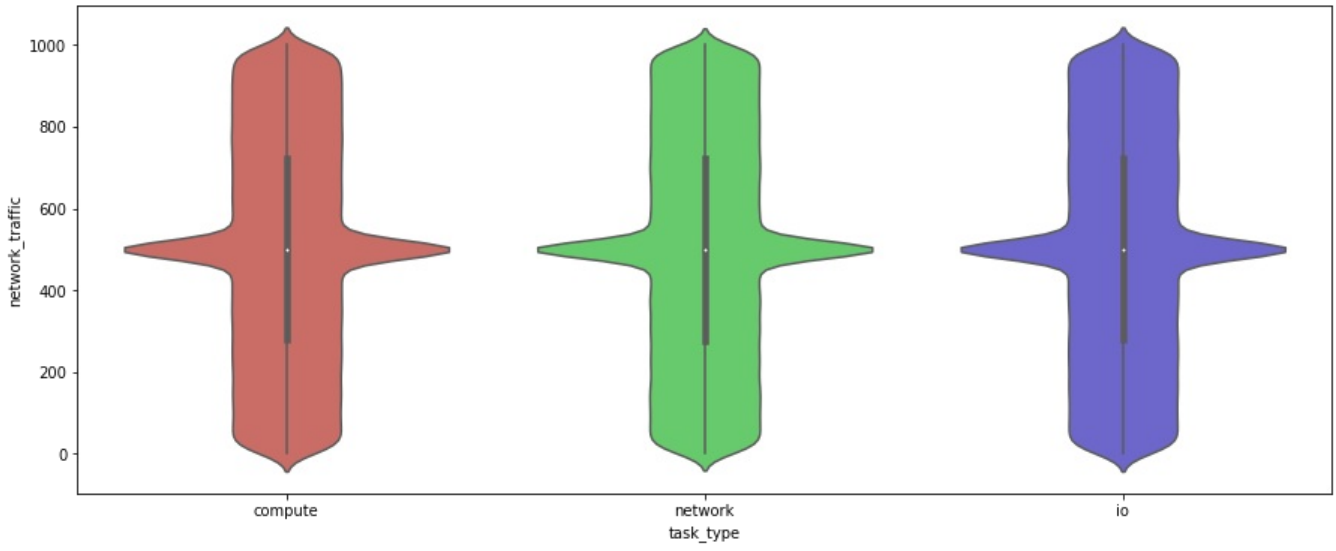


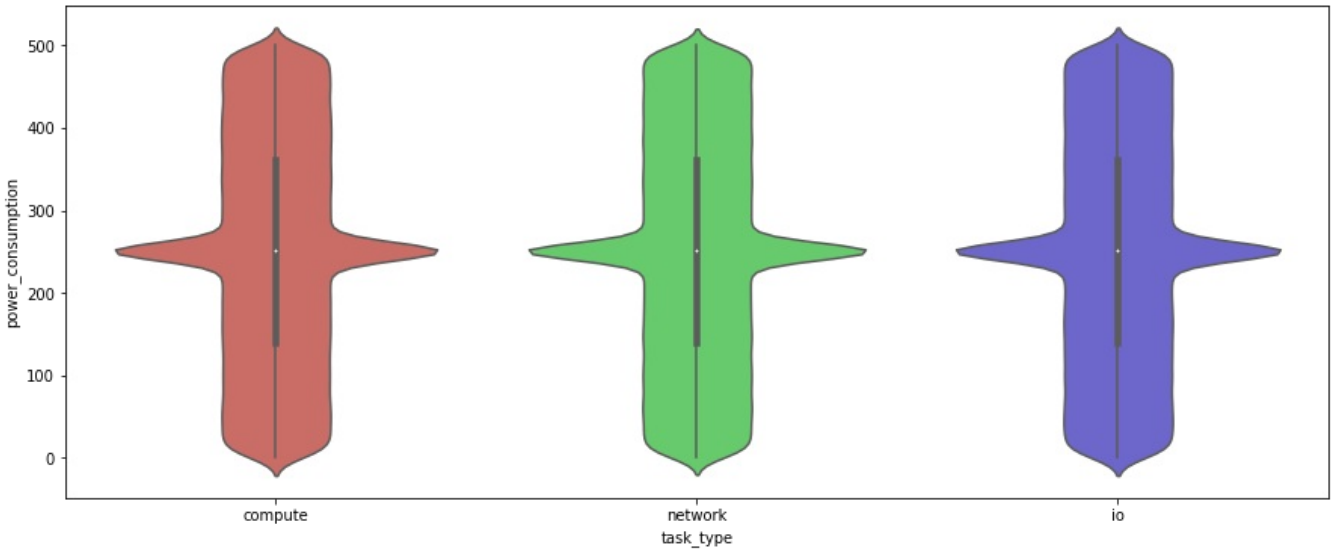


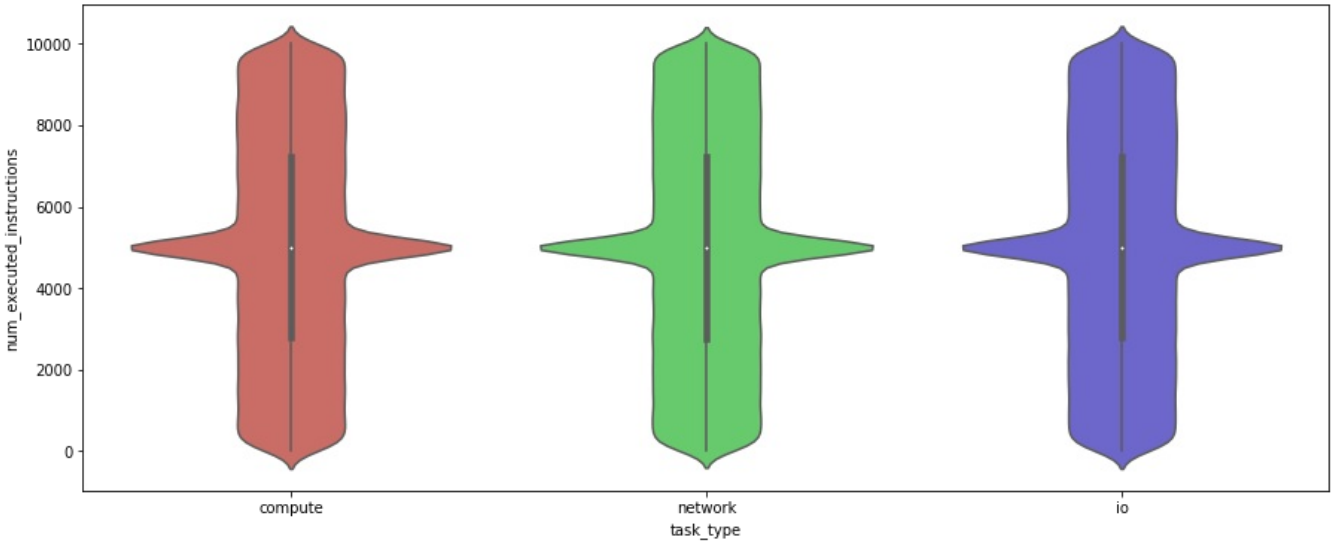
In [42]:
```
for i in categorical_columns:
    for j in numerical_columns:
        plt.figure(figsize=(15,6))
        sns.violinplot(x = df[i], y = df[j], data = df, palette = 'hls')
        plt.show()
```

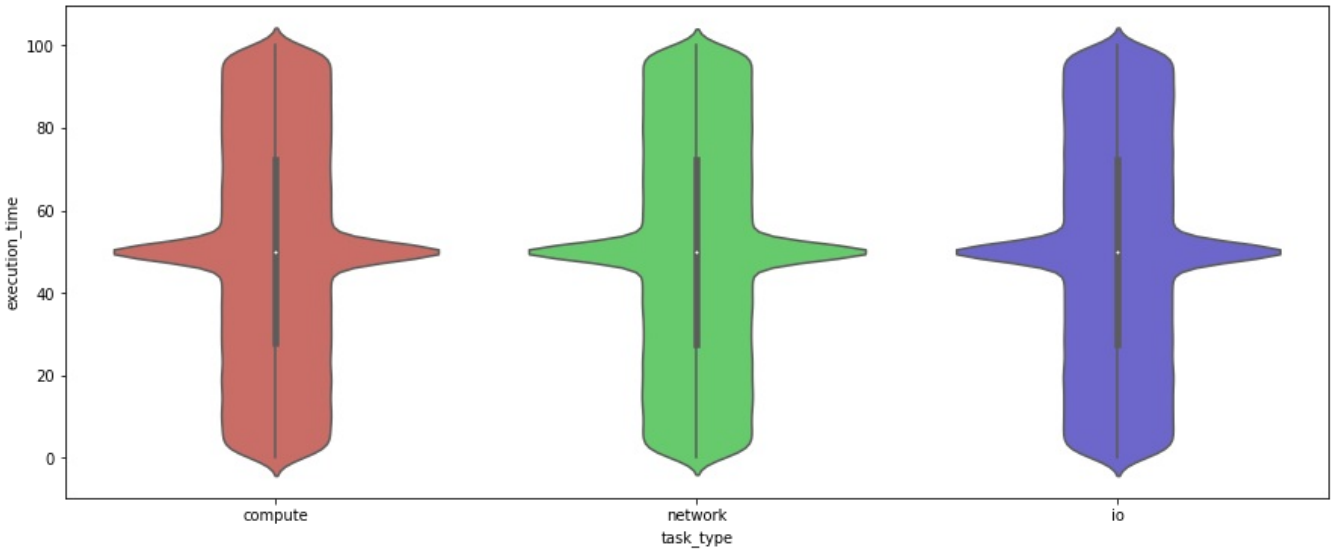

cpu_usage
100
80
60
40
20
0
compute
network
io
task_type

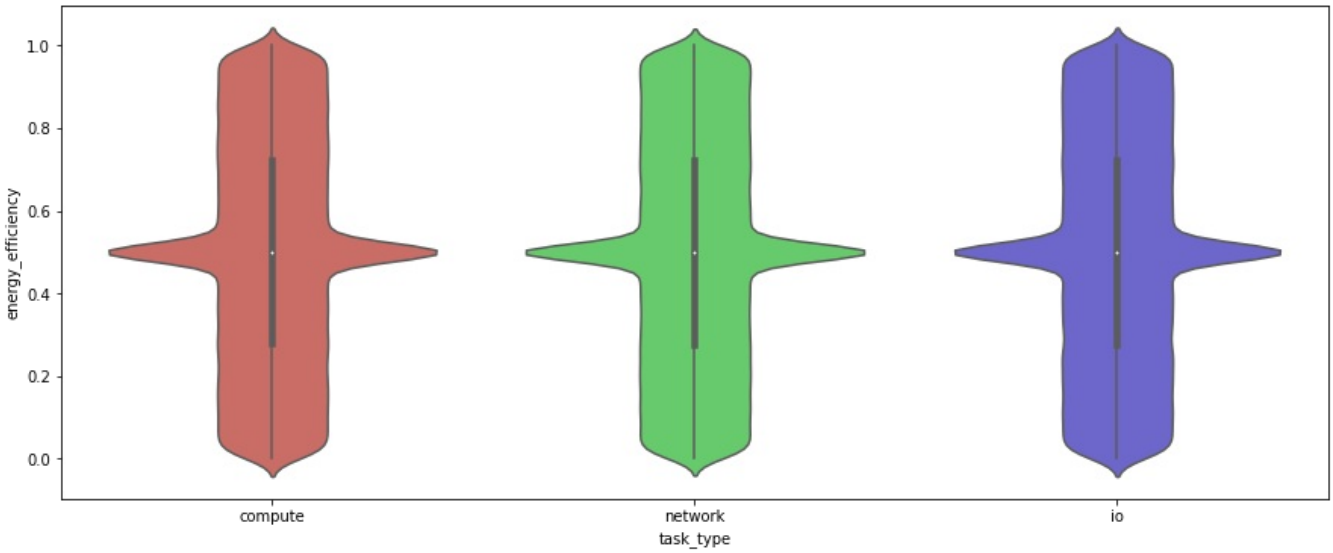

memory_usage
100
80
60
40
20
0
compute
network
io
task_type

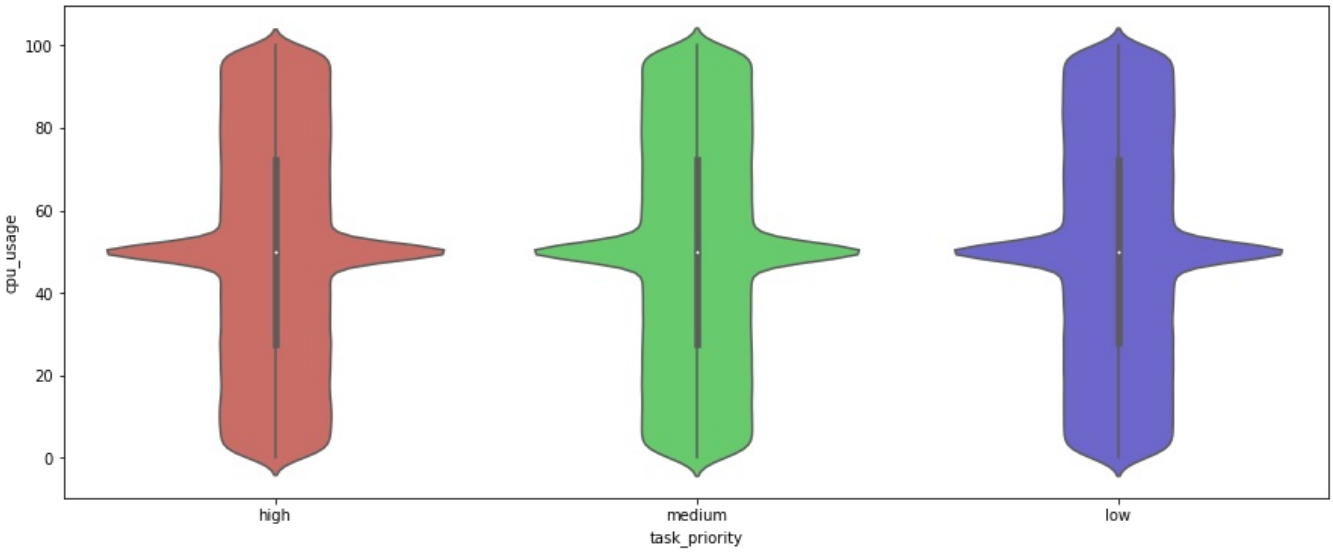

network_traffic
1000
800
600
400
200
0
compute
network
io
task_type

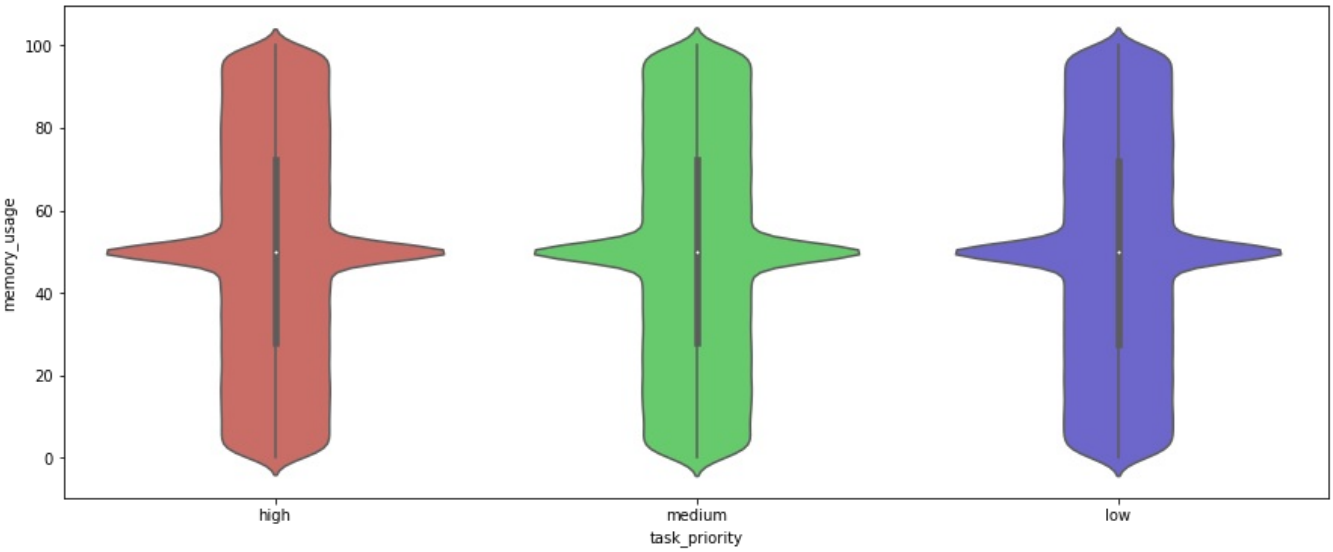

power_consumption
500
400
300
200
100
0
compute
network
io
task_type

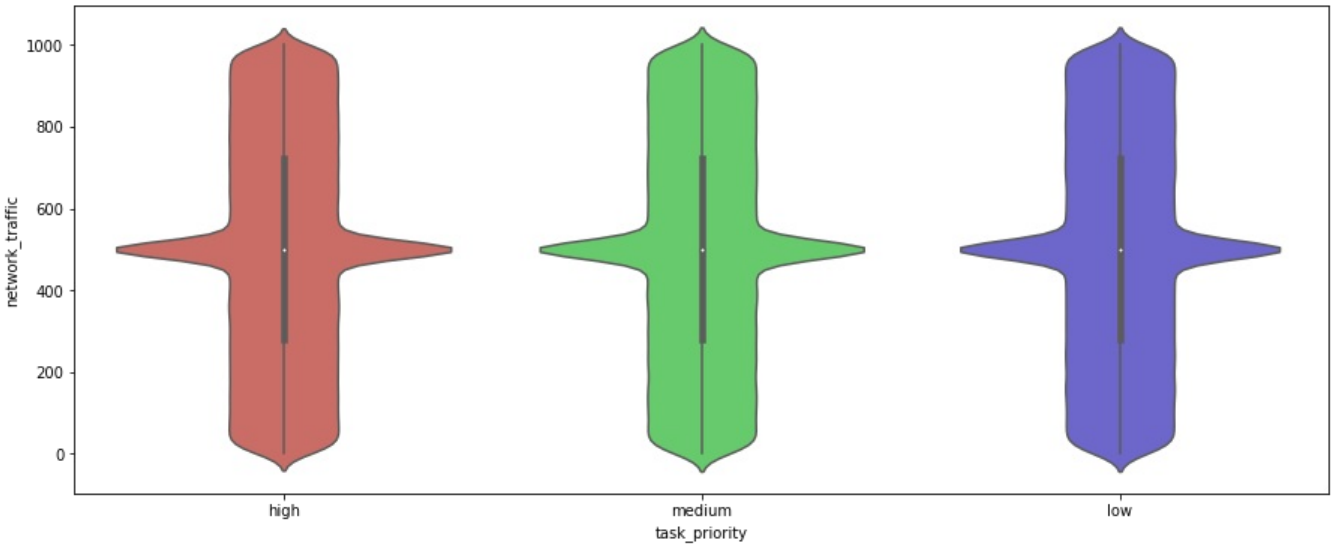

num_executed_instructions
10000
8000
6000
4000
2000
0
compute
network
io
task_type

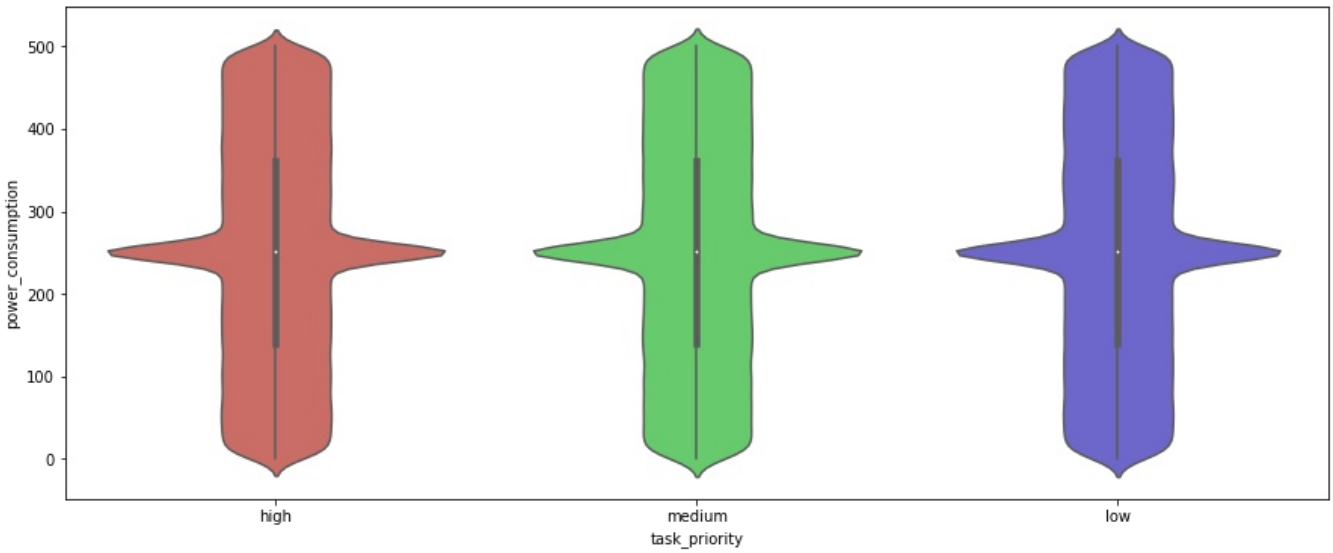

execution_time
100
80
60
40
20
0
compute
network
io
task_type

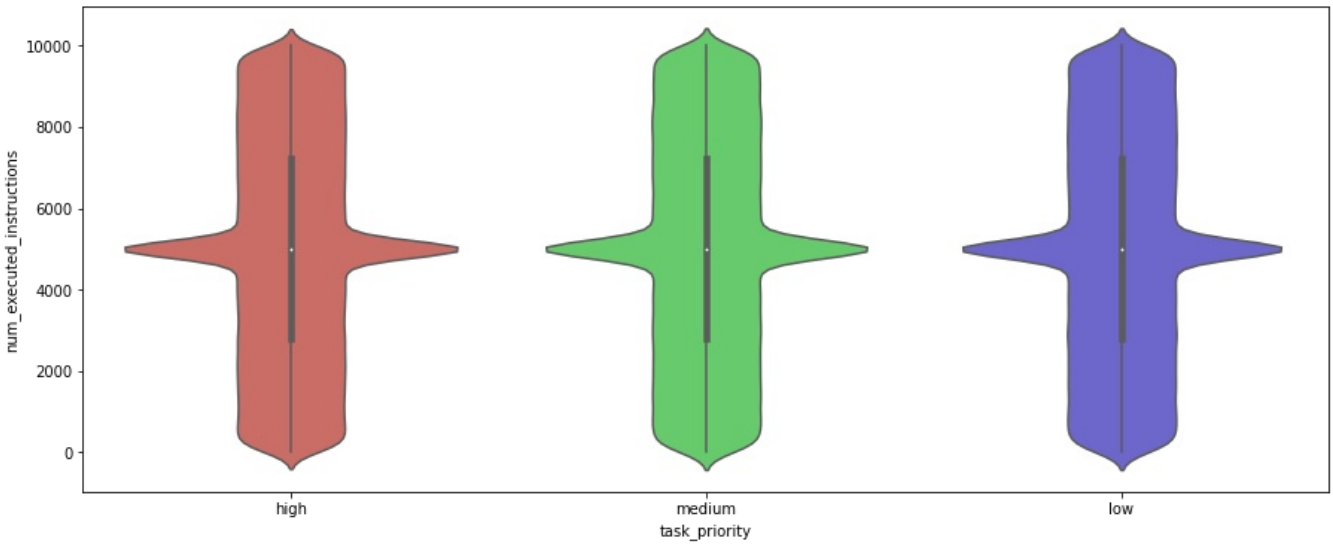

energy_efficiency
1.0
0.8
0.6
0.4
0.2
0.0
compute
network
io
task_type

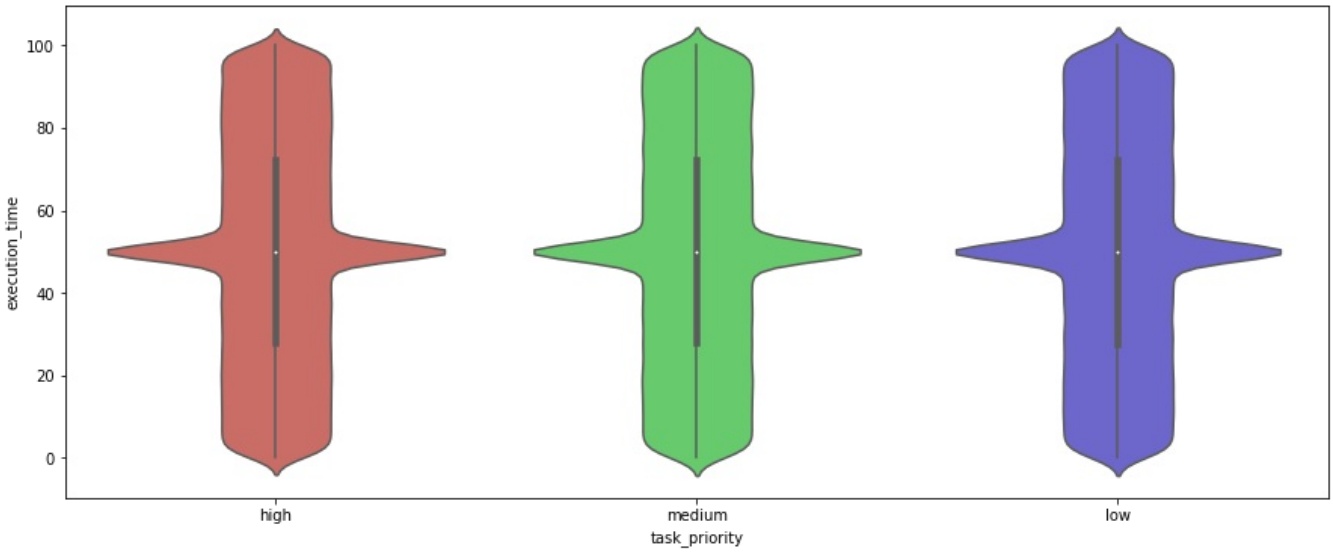

cpu_usage
100
80
60
40
20
0
high
medium
low
task_priority

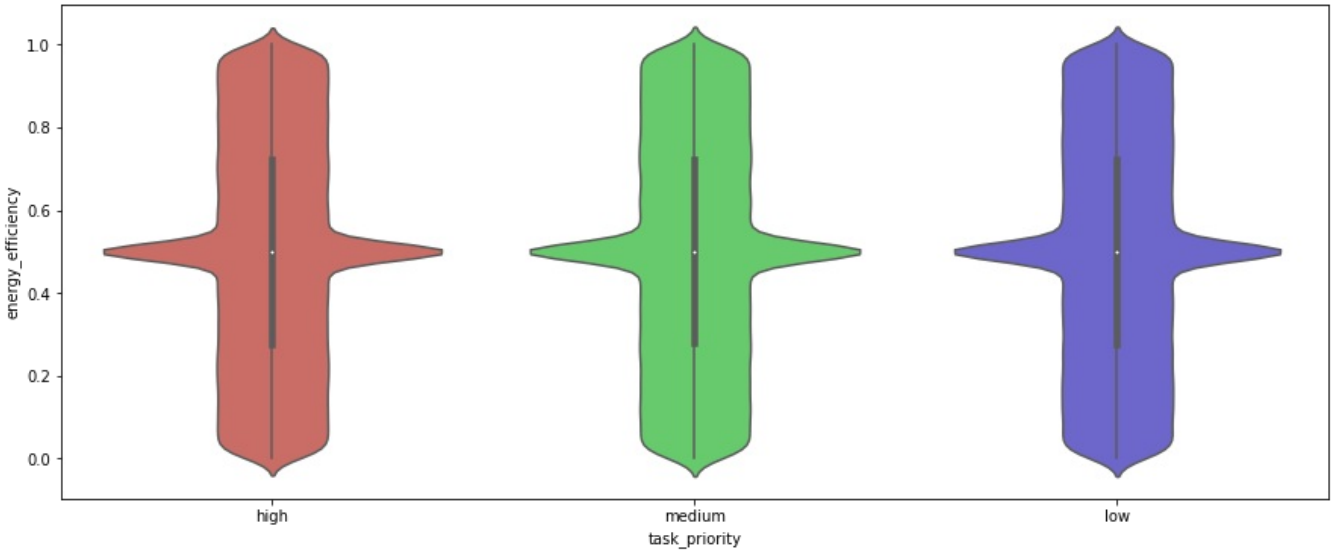

memory_usage
100
80
60
40
20
0
high
medium
low
task_priority

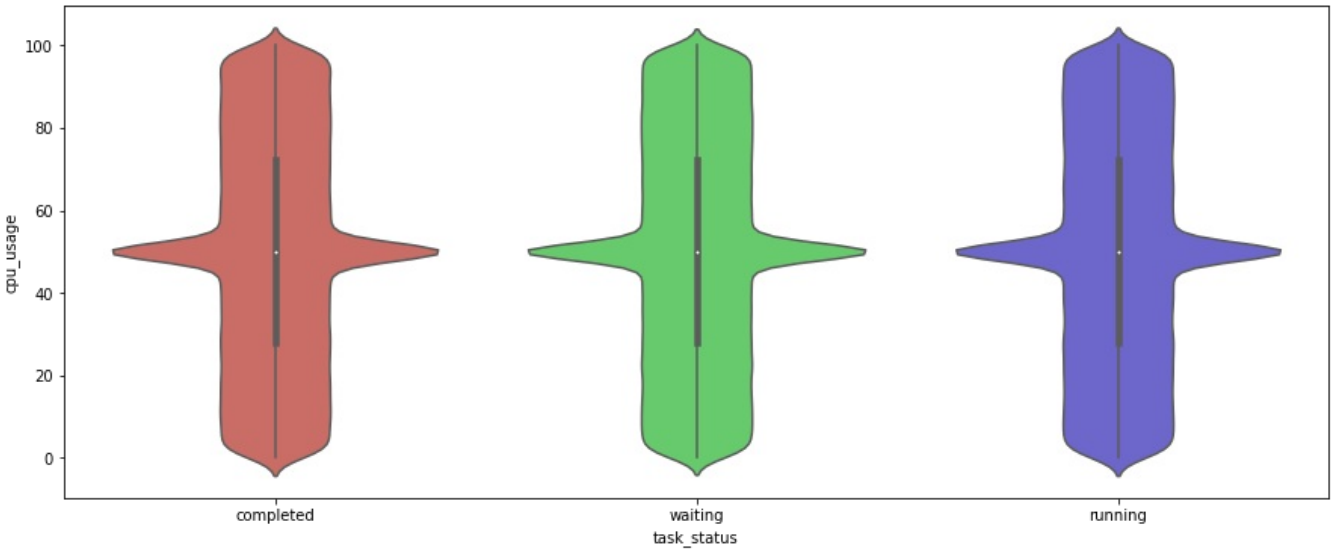

network_traffic
1000
800
600
400
200
0
high
medium
low
task_priority

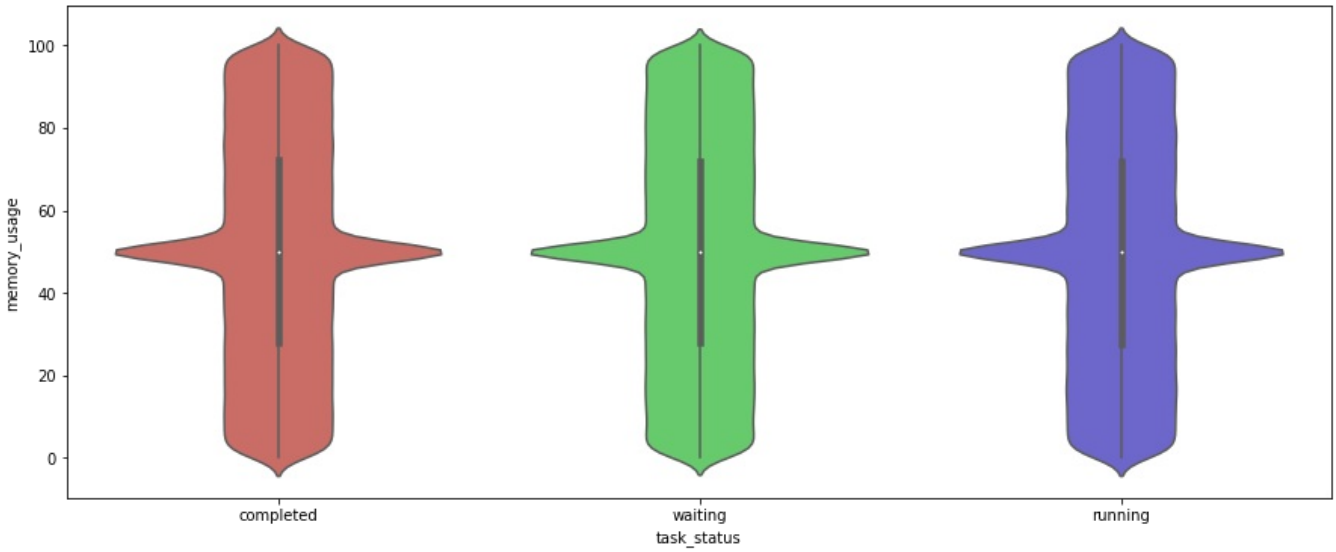

power_consumption
500
400
300
200
100
0
high
medium
low
task_priority

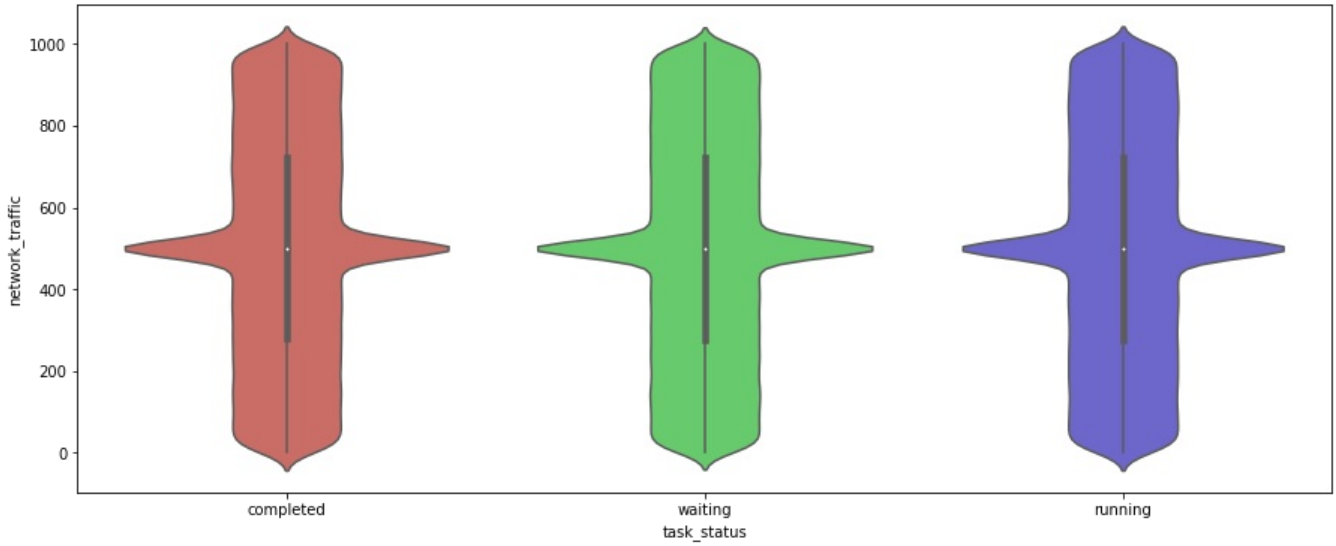

num_executed_instructions
10000
8000
6000
4000
2000
0
high
medium
low
task_priority

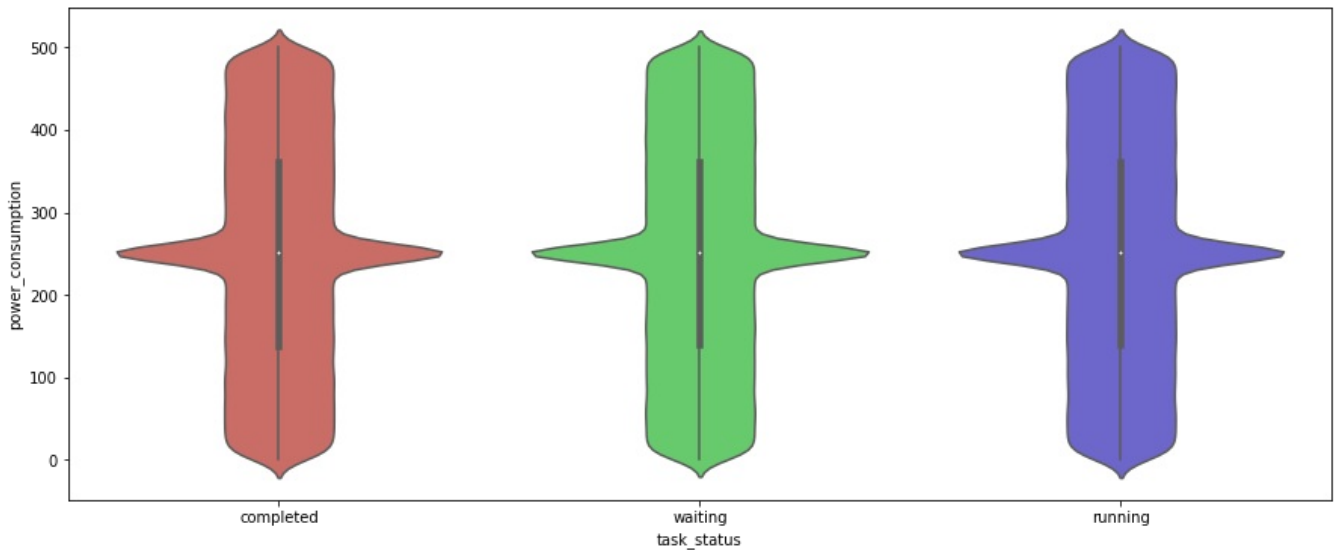

execution_time
100
80
60
40
20
0
high
medium
low
task_priority

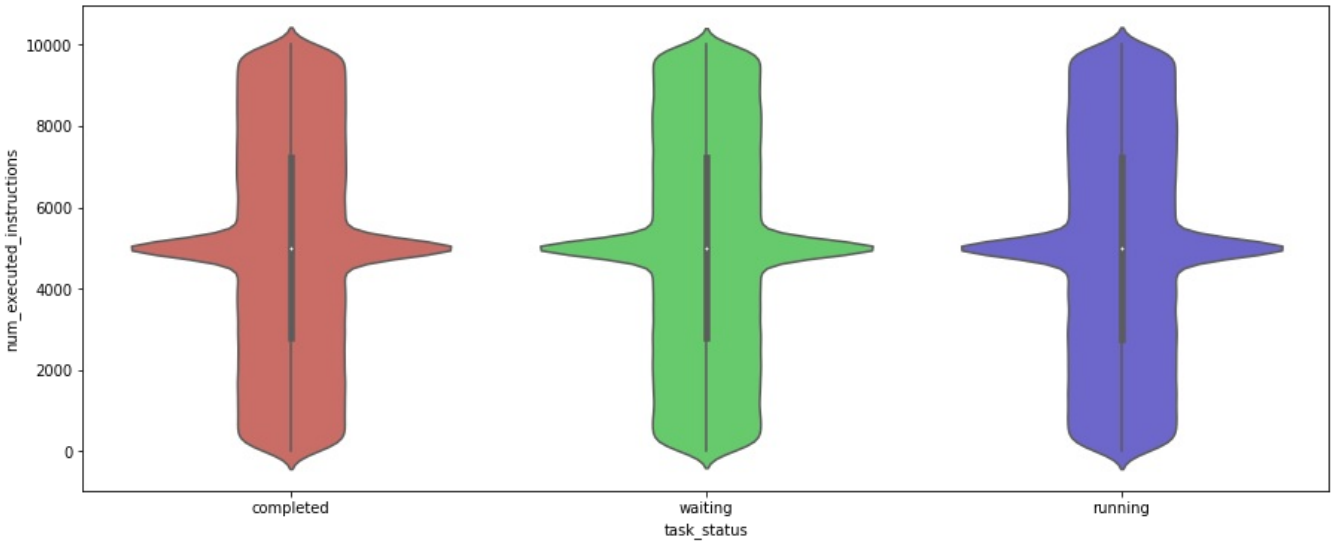

energy_efficiency
1.0
0.8
0.6
0.4
0.2
0.0
high
medium
low
task_priority

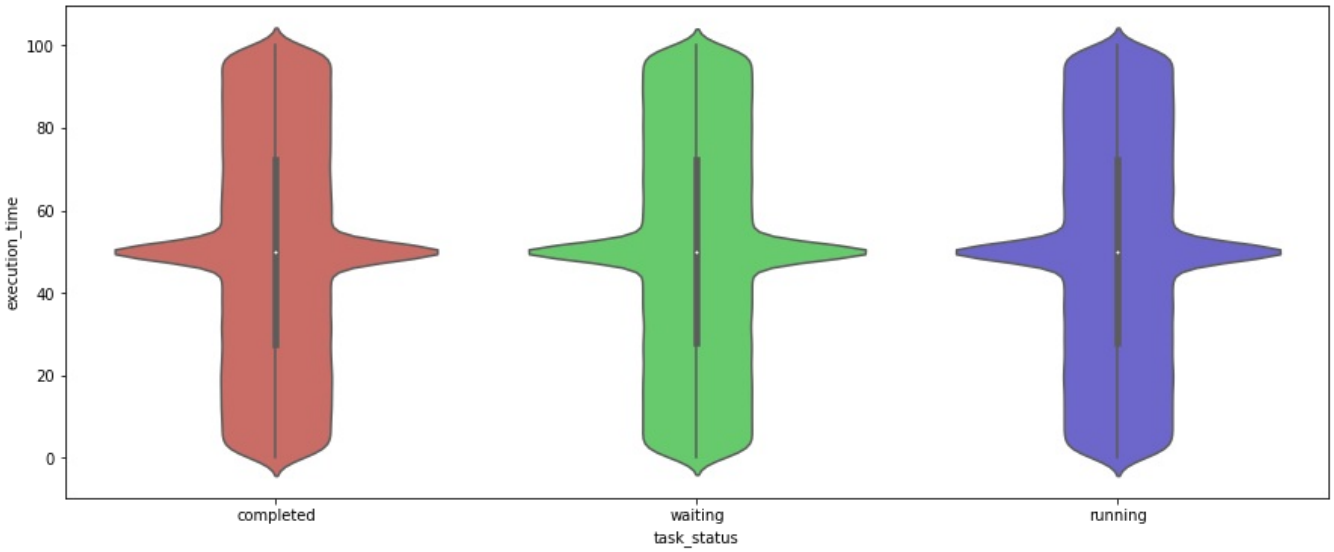

cpu_usage
100
80
60
40
20
0
completed
waiting
running
task_status

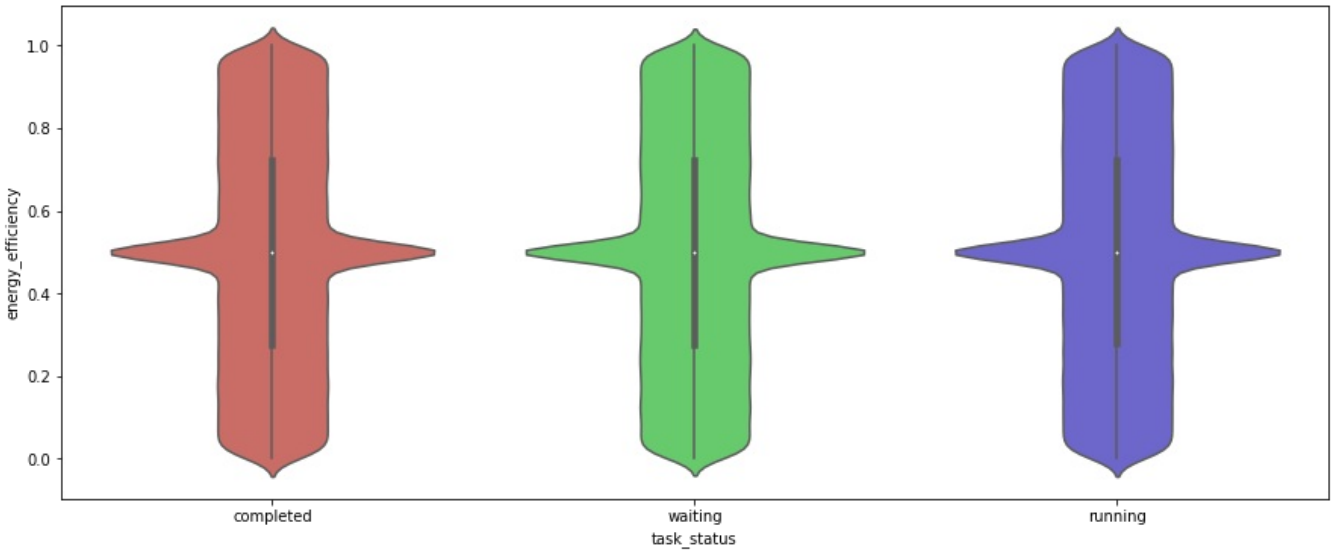

memory_usage
100
80
60
40
20
0
completed
waiting
running
task_status


network_traffic
1000
800
600
400
200
0
completed
waiting
running
task_status

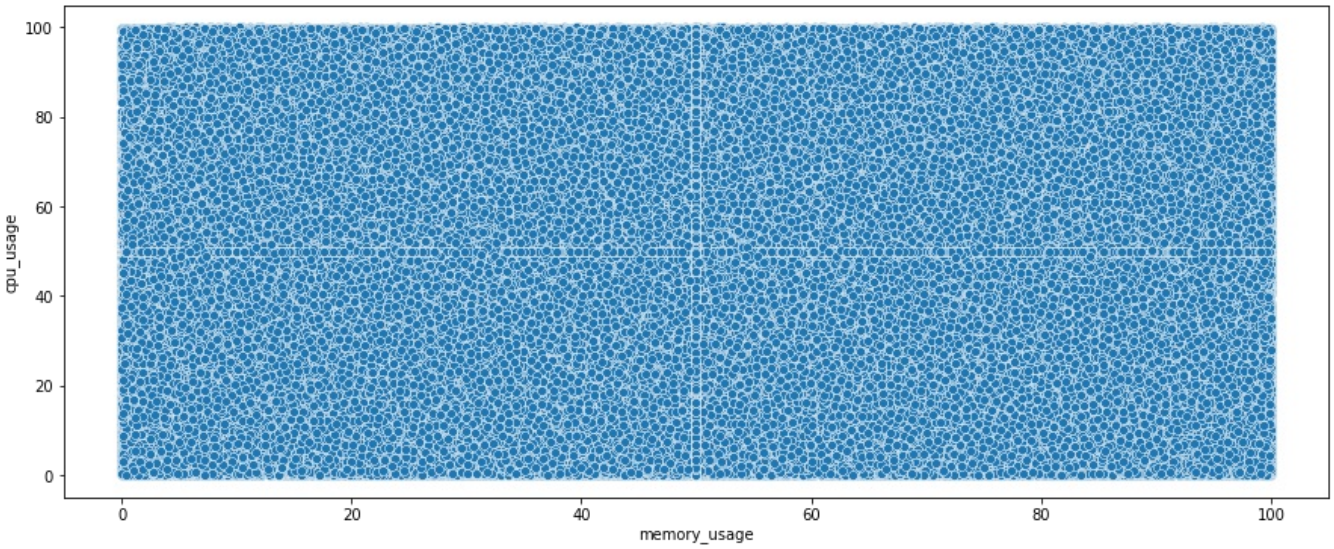

power_consumption
500
400
300
200
100
0
completed
waiting
running
task_status

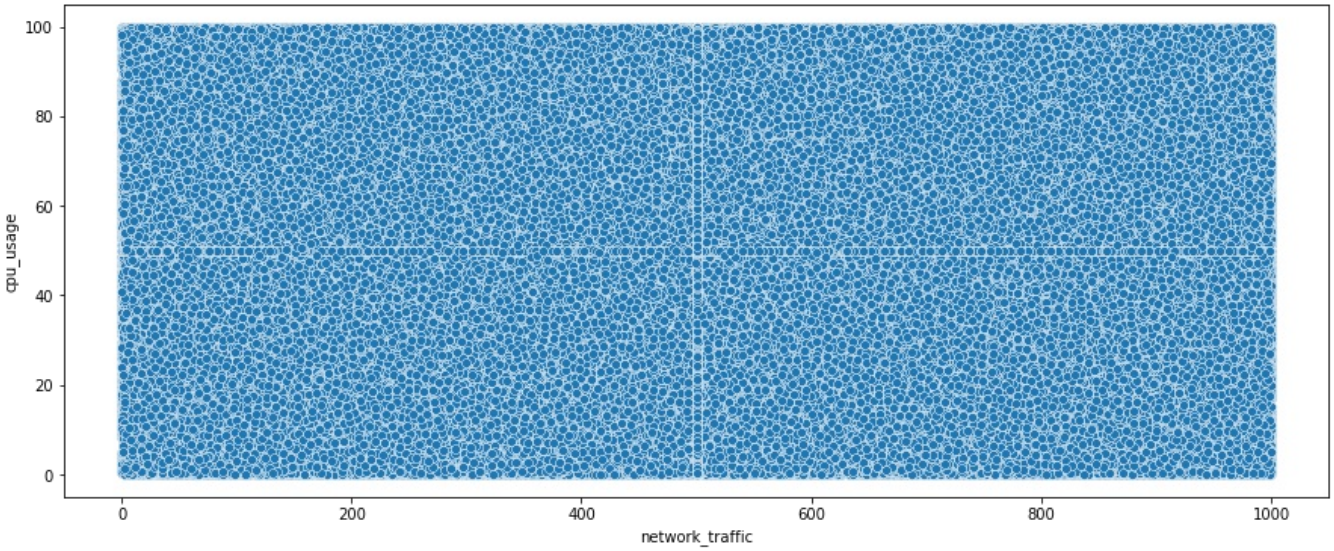


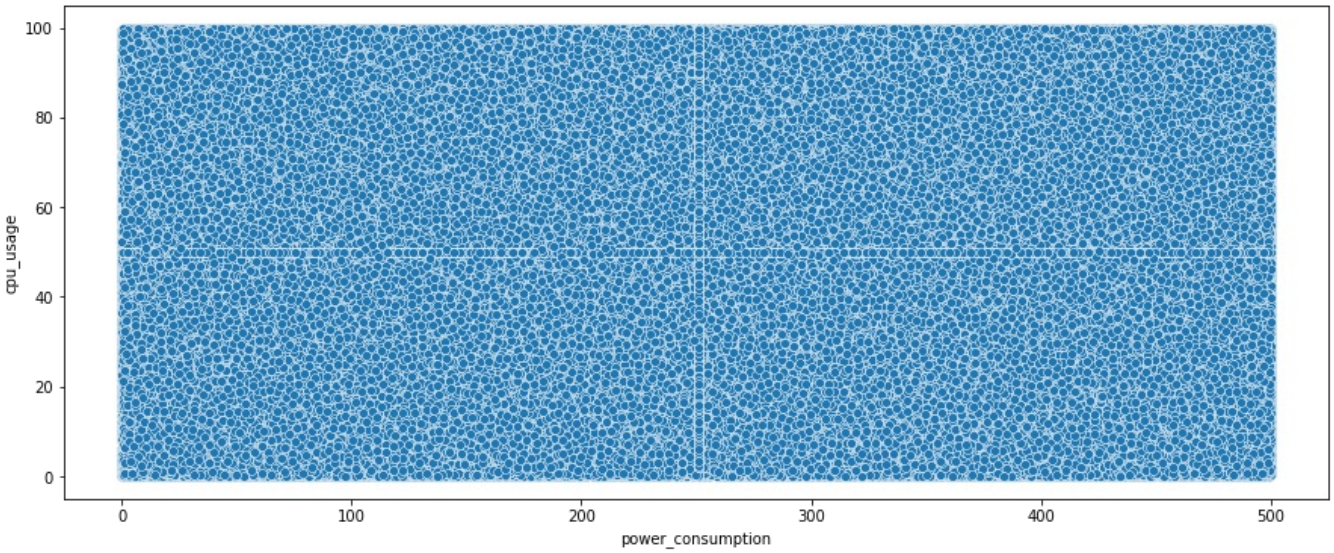


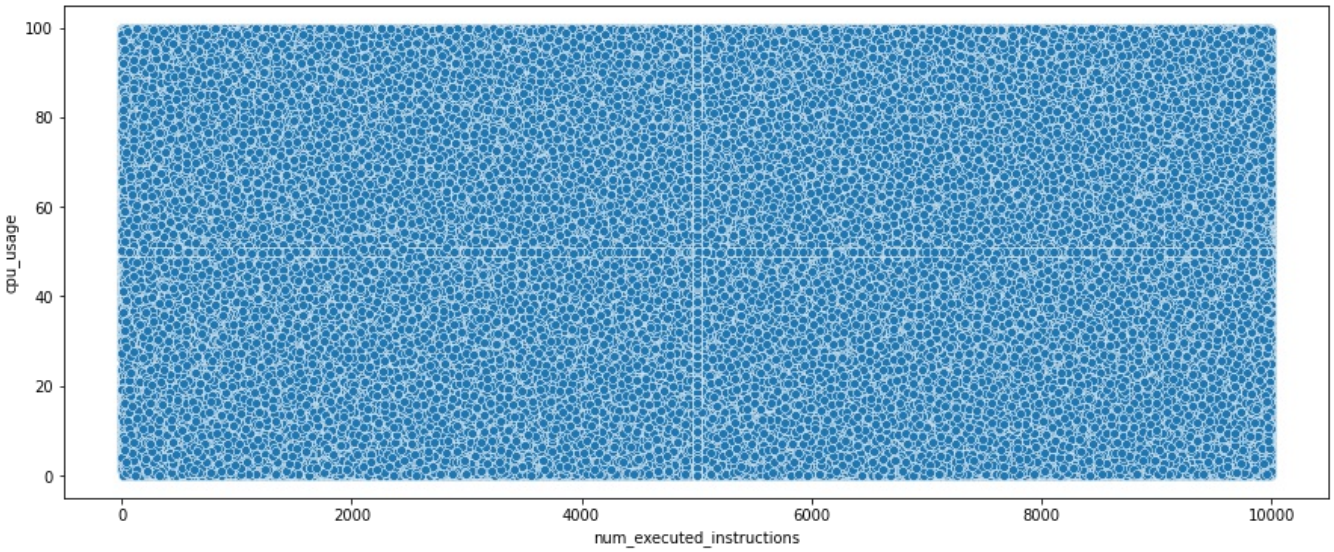


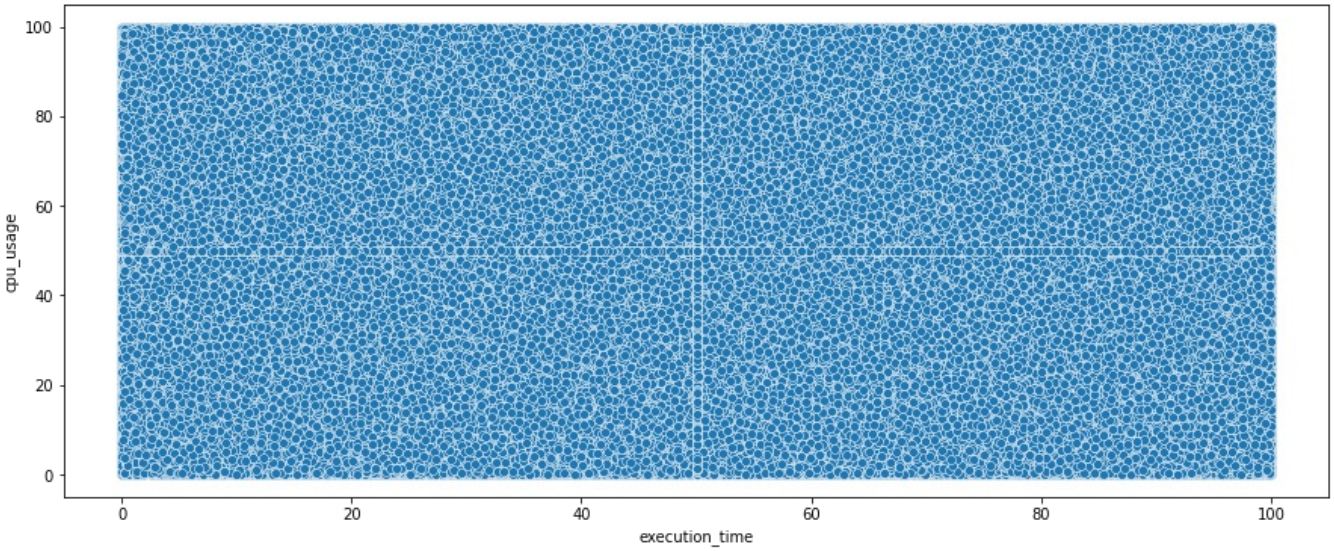
In [43]:
```
for i in numerical_columns:
    for j in numerical_columns:
        if i != j:
            plt.figure(figsize=(15,6))
            sns.scatterplot(x = df[j], y = df[i], data = df, palette = 'hls')
            plt.show()
```

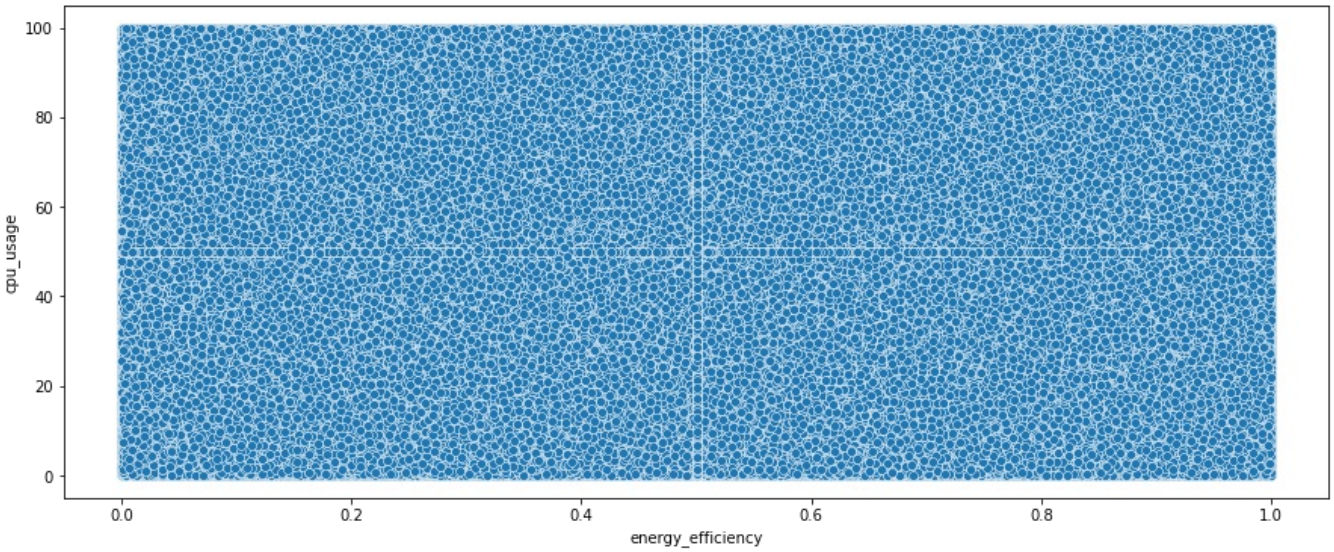

cpu_usage
memory_usage
0
20
40
60
80
100

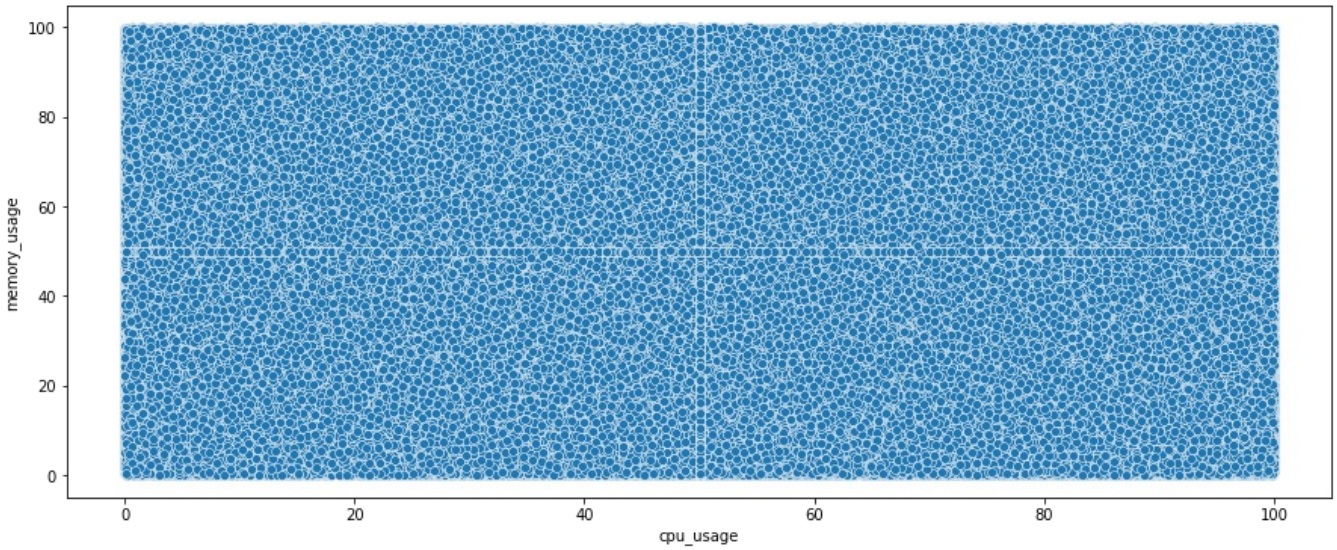

cpu_usage
network_traffic
0
200
400
600
800
1000

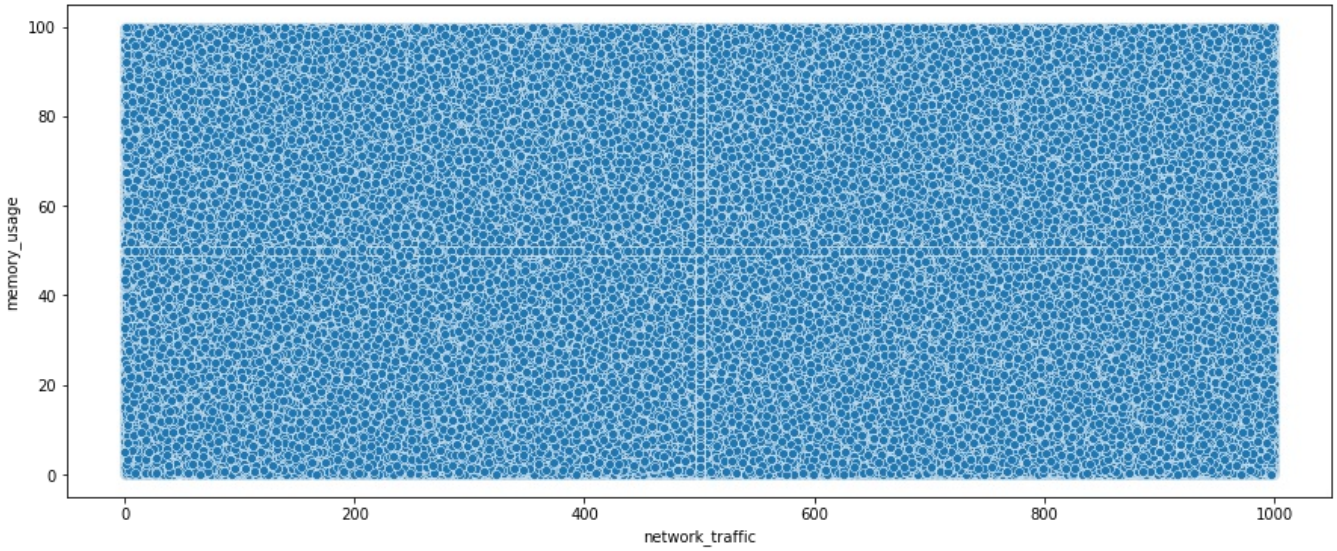

cpu_usage
power_consumption
0
100
200
300
400
500

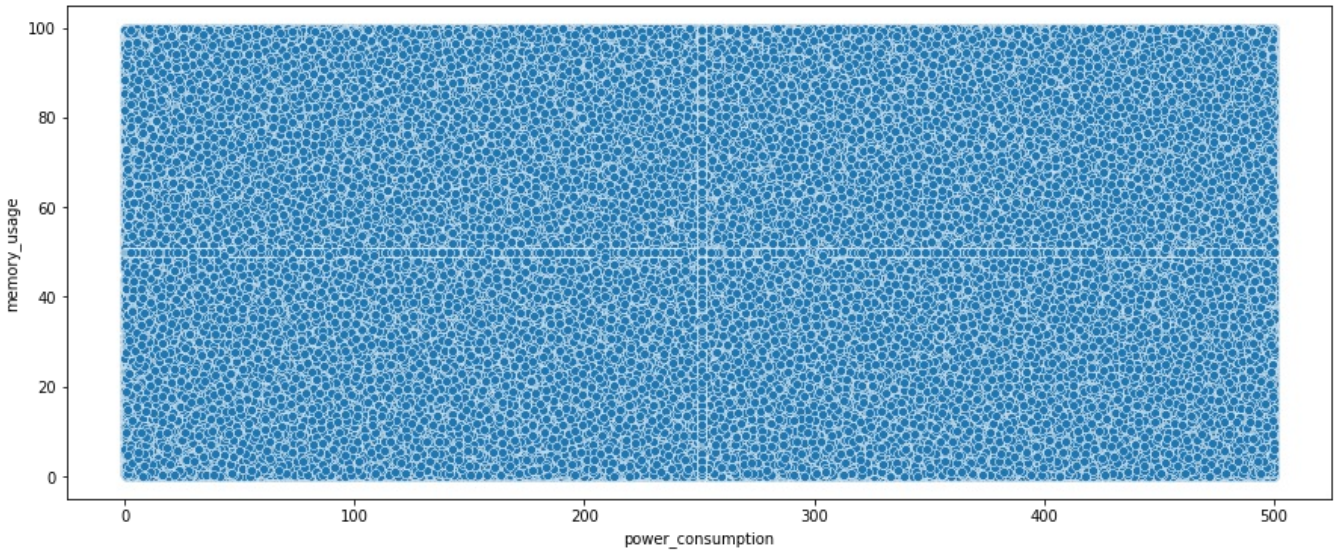

cpu_usage
num_executed_instructions
0
20
40
60
80
100
0
2000
4000
6000
8000
10000

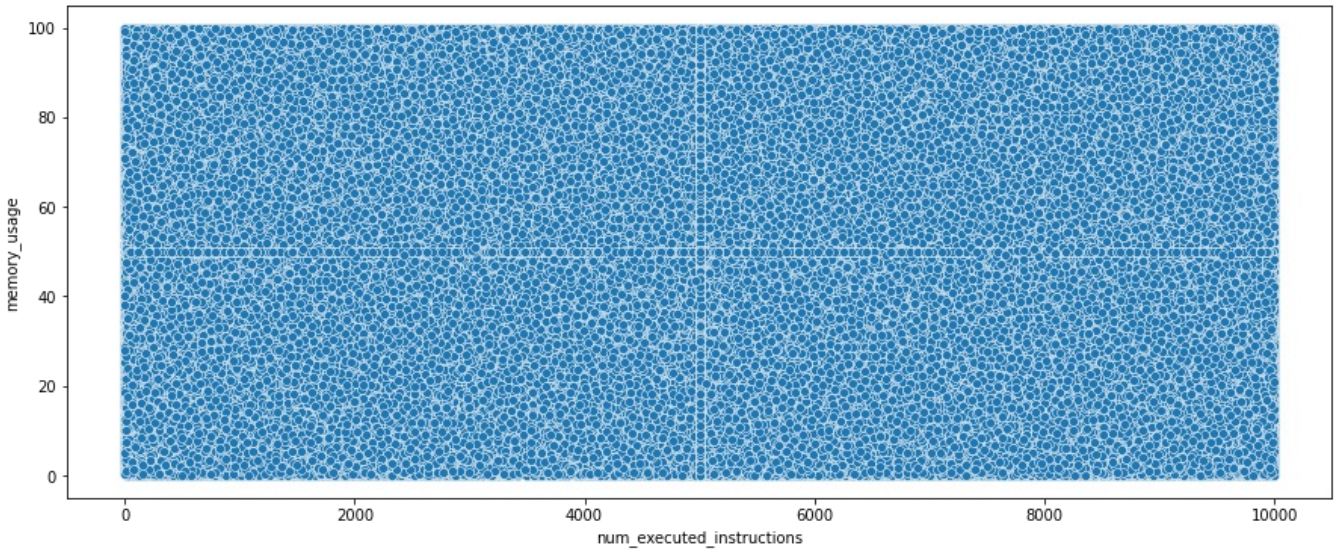

cpu_usage
execution_time
0
20
40
60
80
100

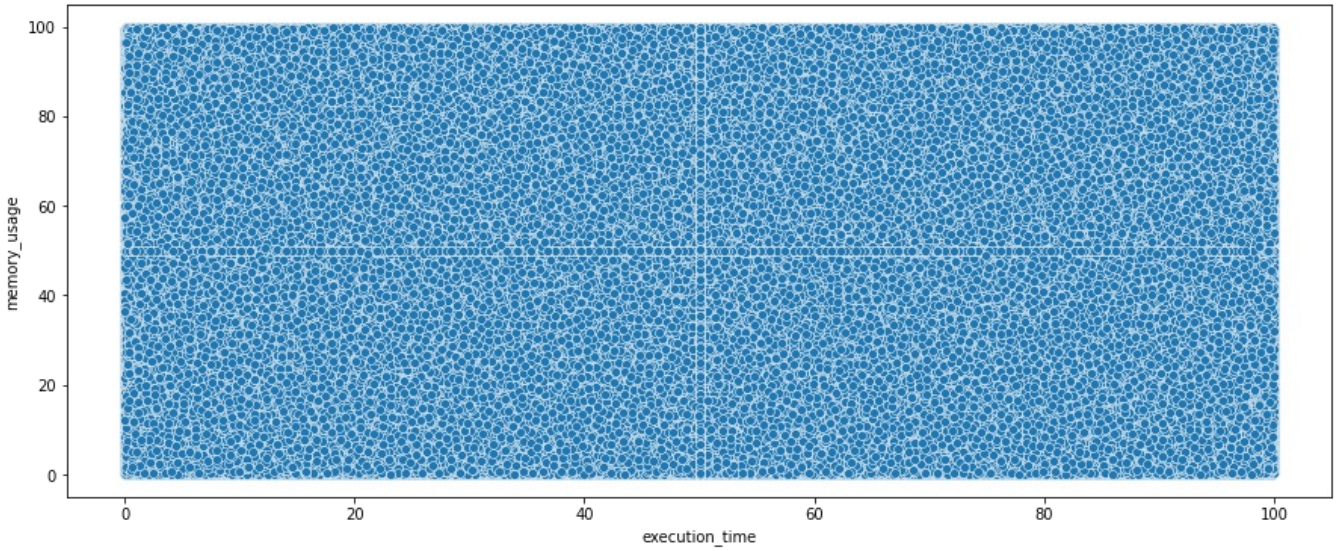

cpu_usage
energy_efficiency
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

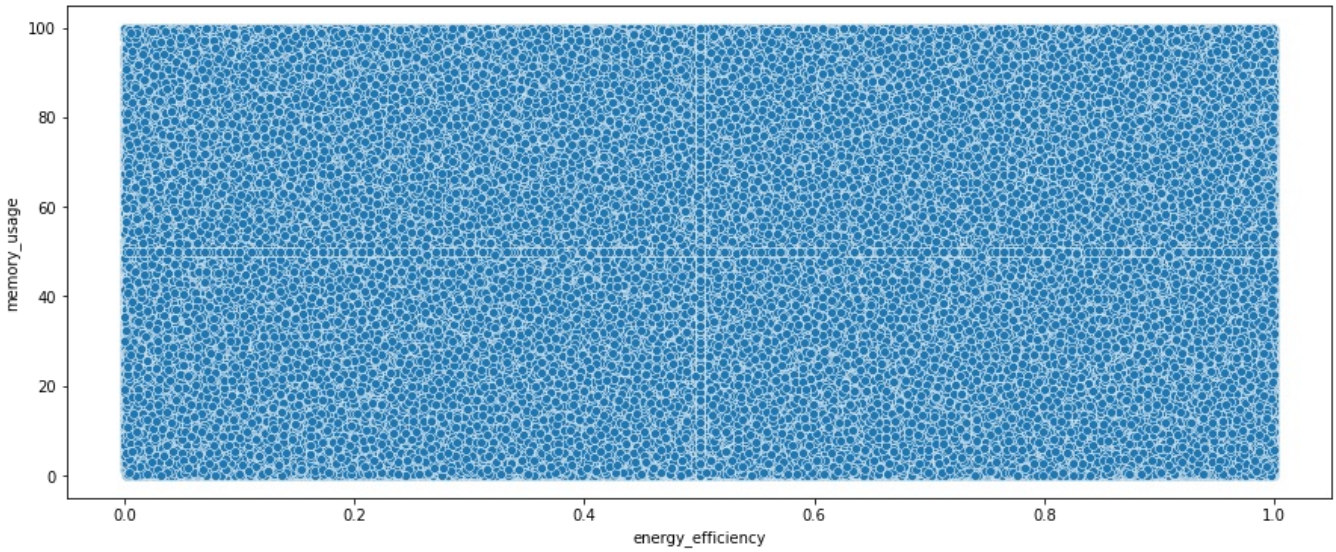

memory_usage
cpu_usage
0
20
40
60
80
100

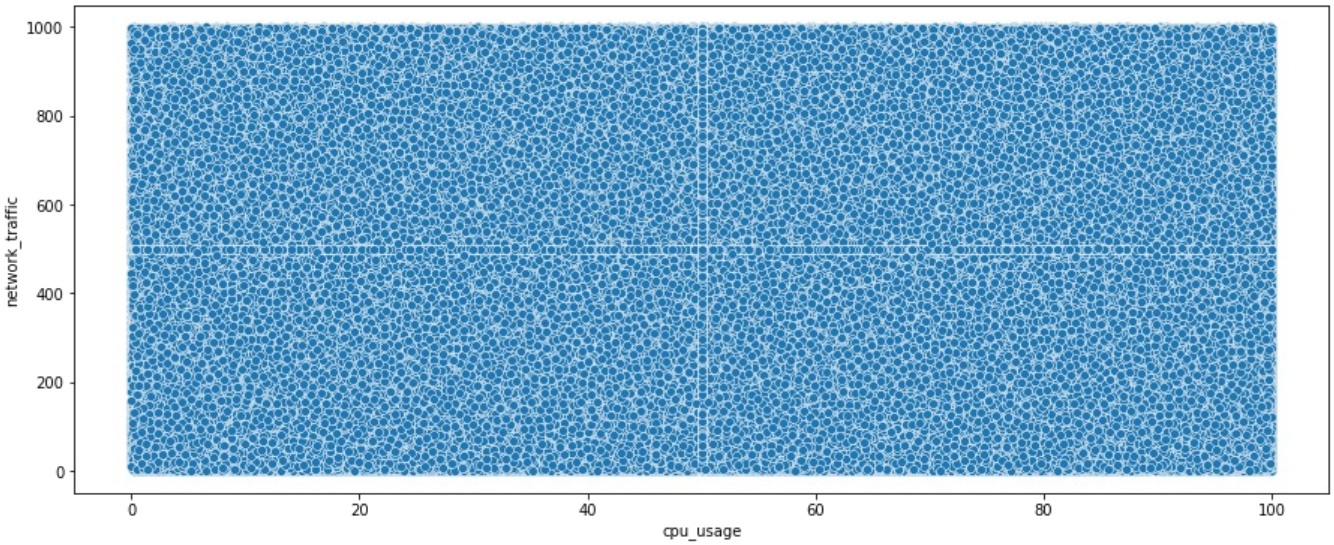

memory_usage
network_traffic
0
200
400
600
800
1000

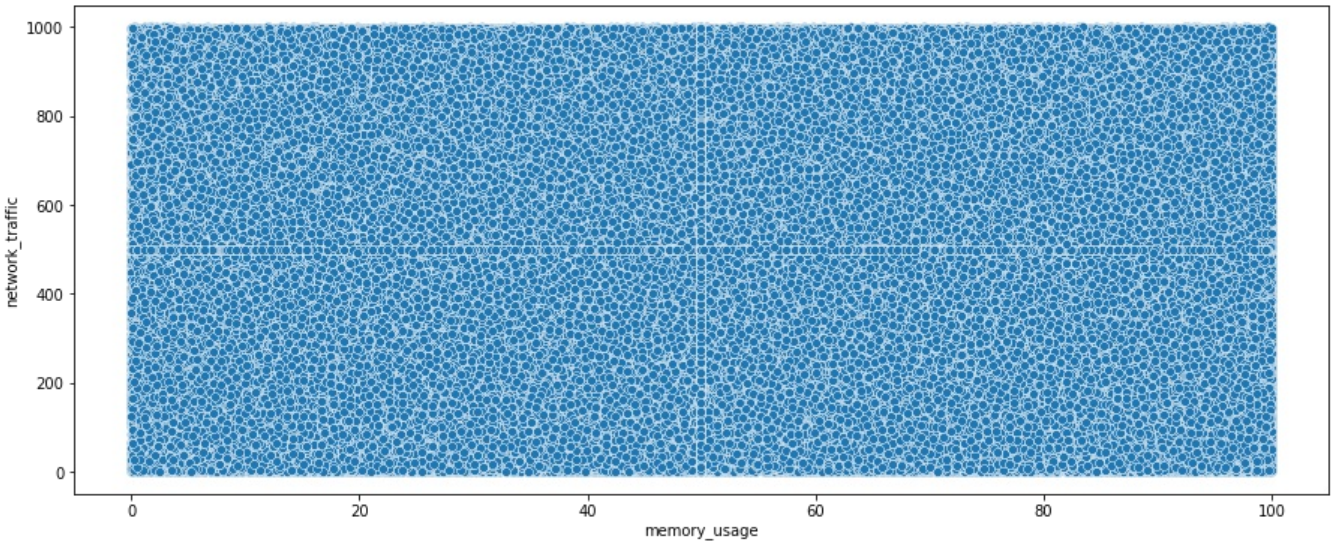

memory_usage
power_consumption
0
100
200
300
400
500

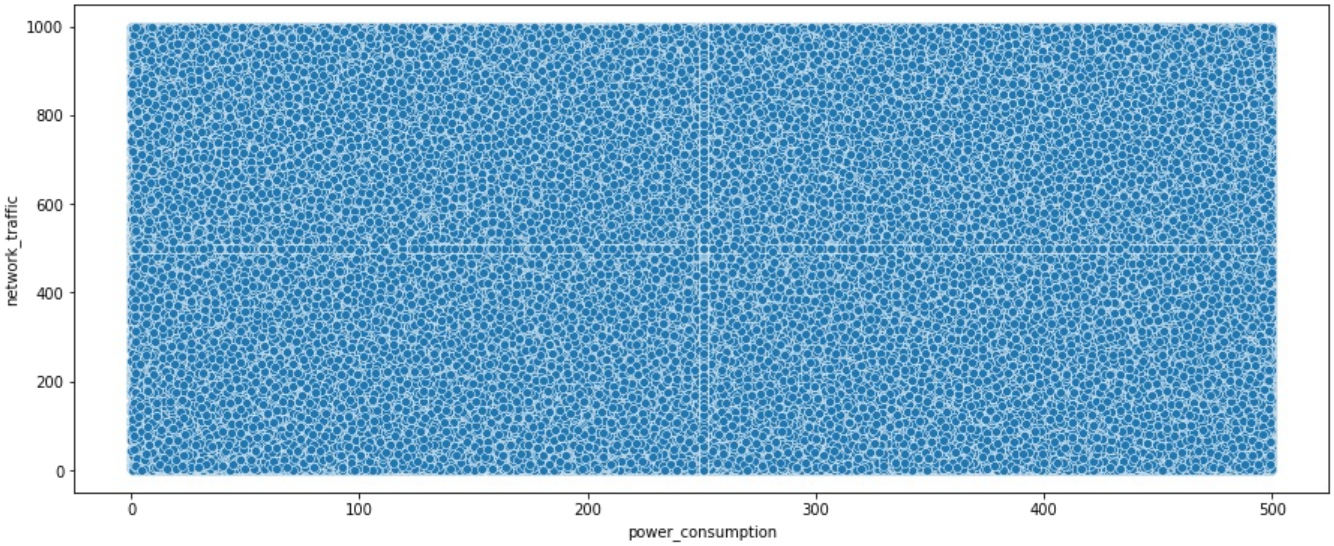

memory_usage
num_executed_instructions
0
20
40
60
80
100
0
2000
4000
6000
8000
10000

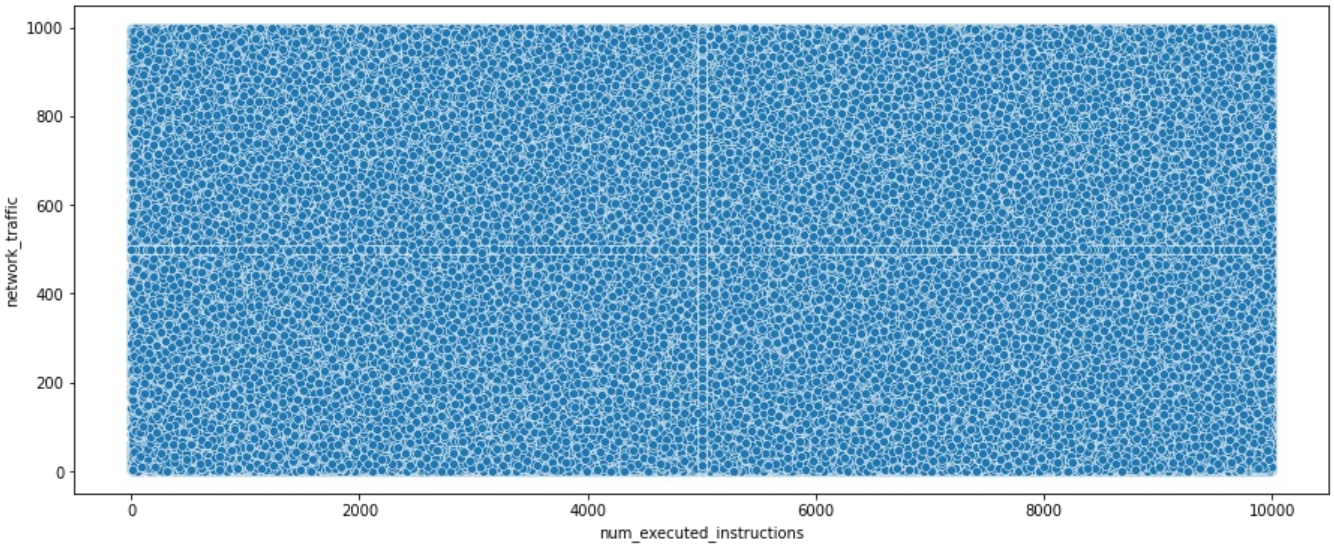

memory_usage
execution_time
0
20
40
60
80
100

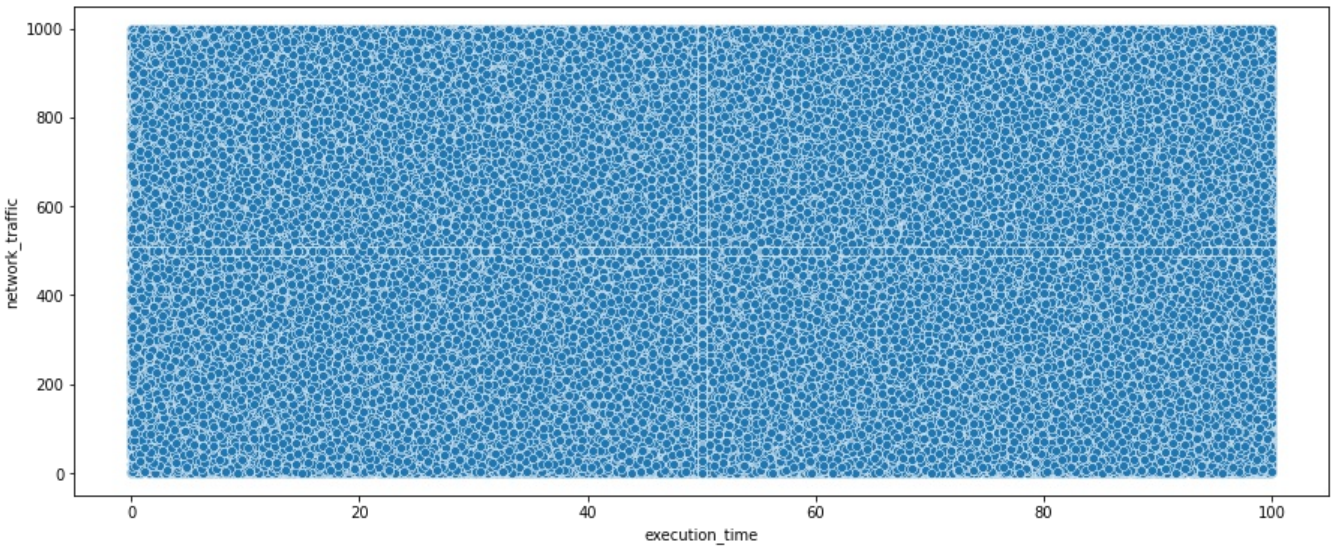

memory_usage
energy_efficiency
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

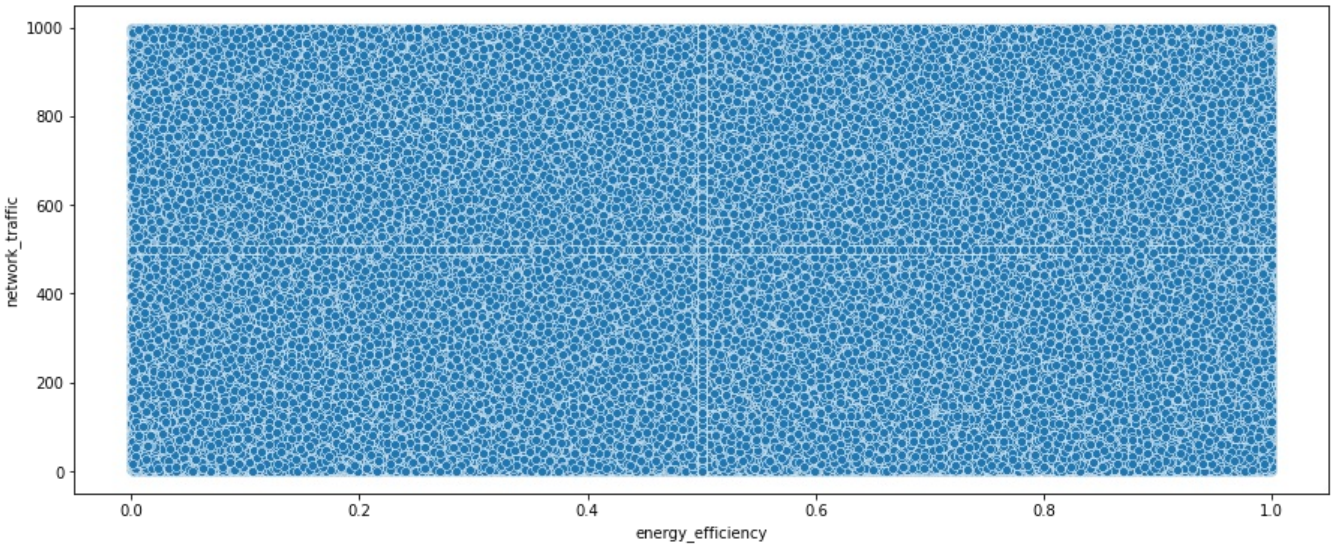

network_traffic
cpu_usage
0
200
400
600
800
1000
0
20
40
60
80
100

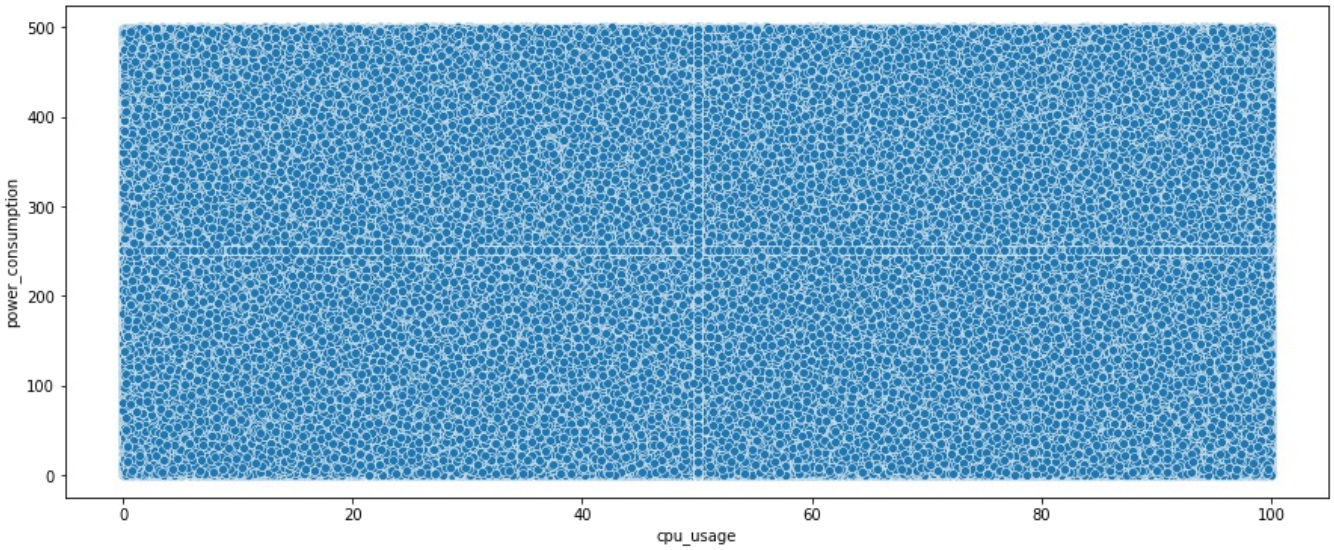

network_traffic
memory_usage
0
200
400
600
800
1000
0
20
40
60
80
100

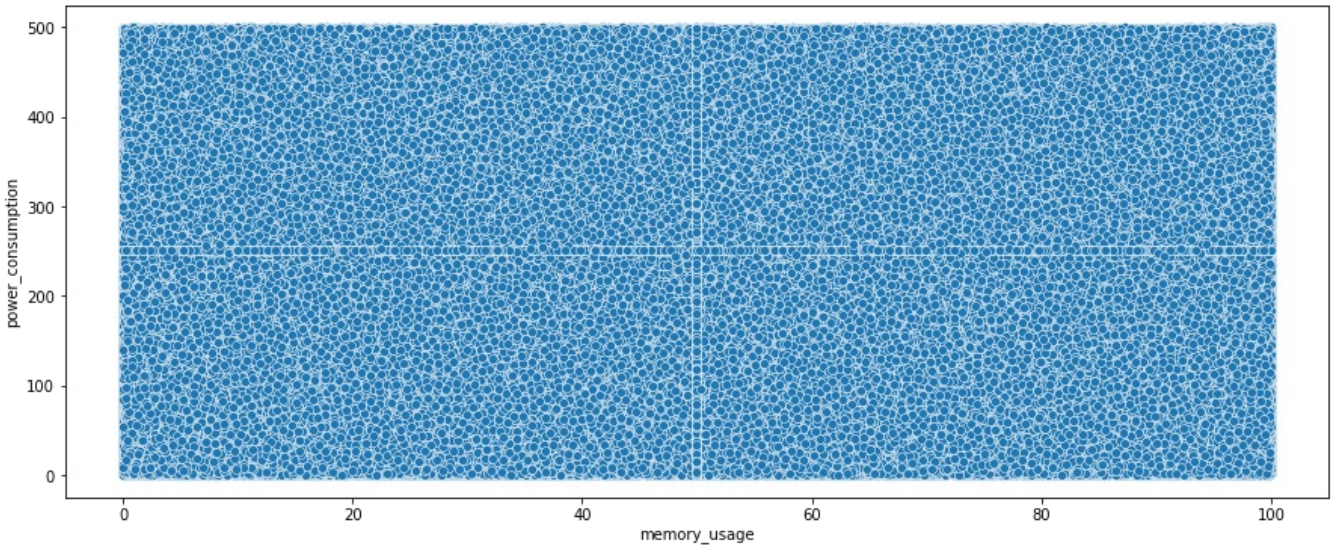

network_traffic
power_consumption
0
200
400
600
800
1000
0
100
200
300
400
500

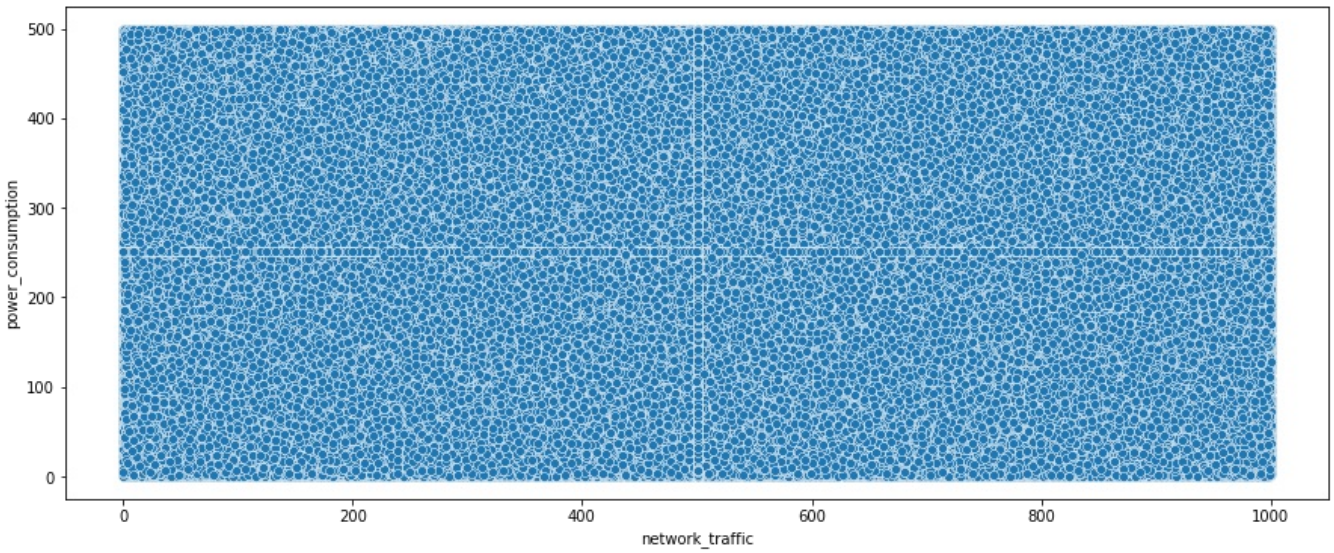

network_traffic
num_executed_instructions
0
200
400
600
800
1000
0
2000
4000
6000
8000
10000

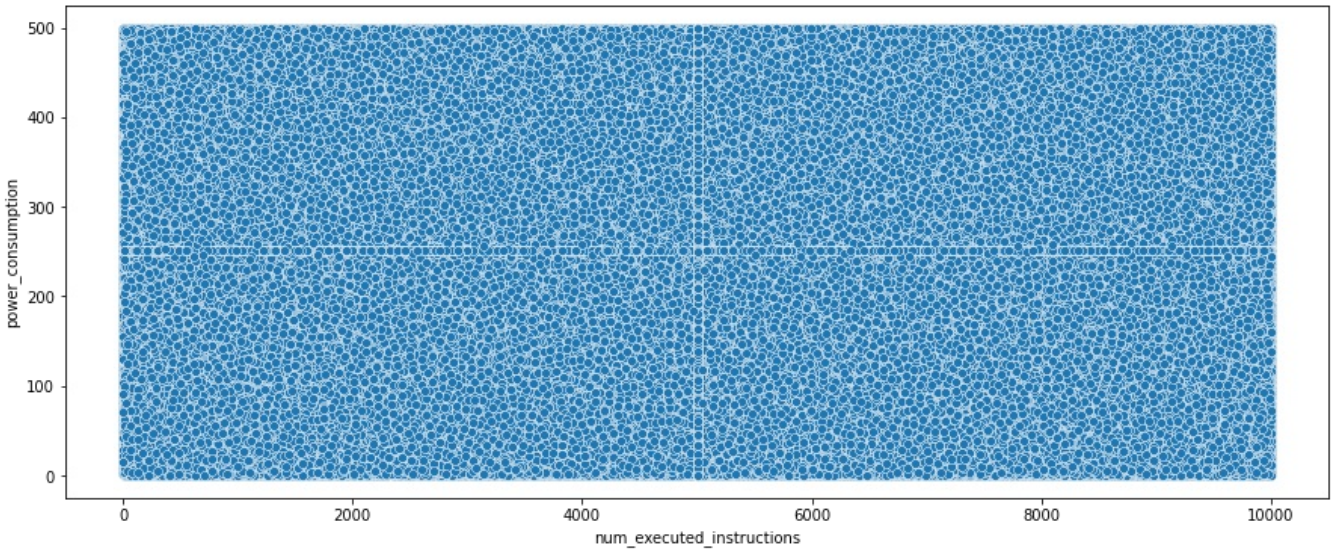

network_traffic
execution_time
0
200
400
600
800
1000
0
20
40
60
80
100

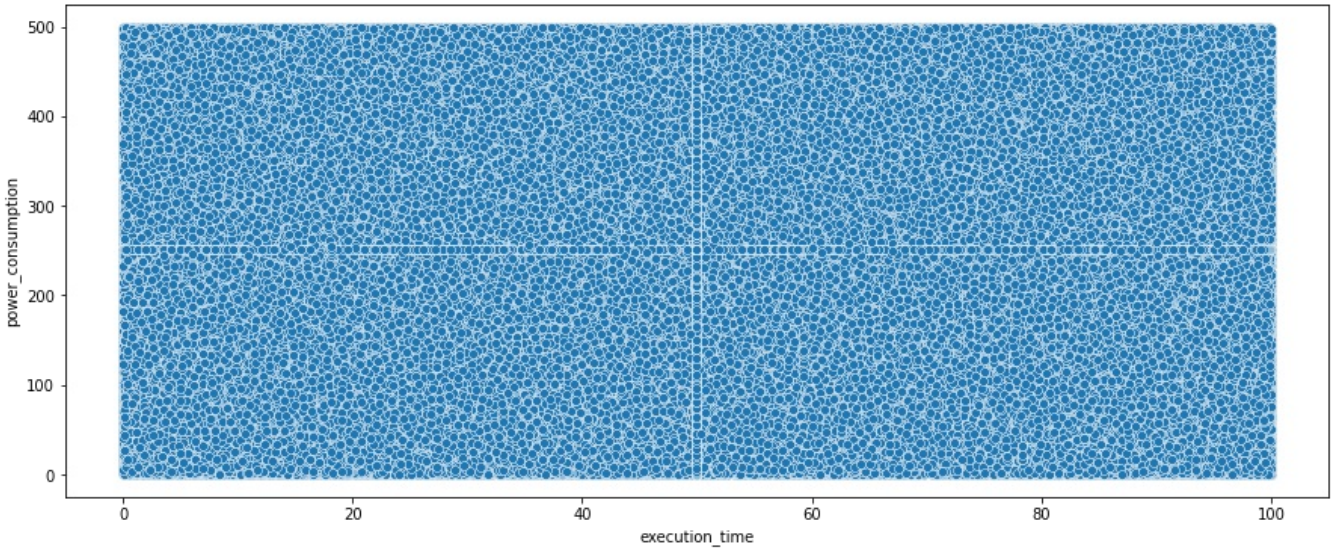

network_traffic
energy_efficiency
0
200
400
600
800
1000
0.0
0.2
0.4
0.6
0.8
1.0

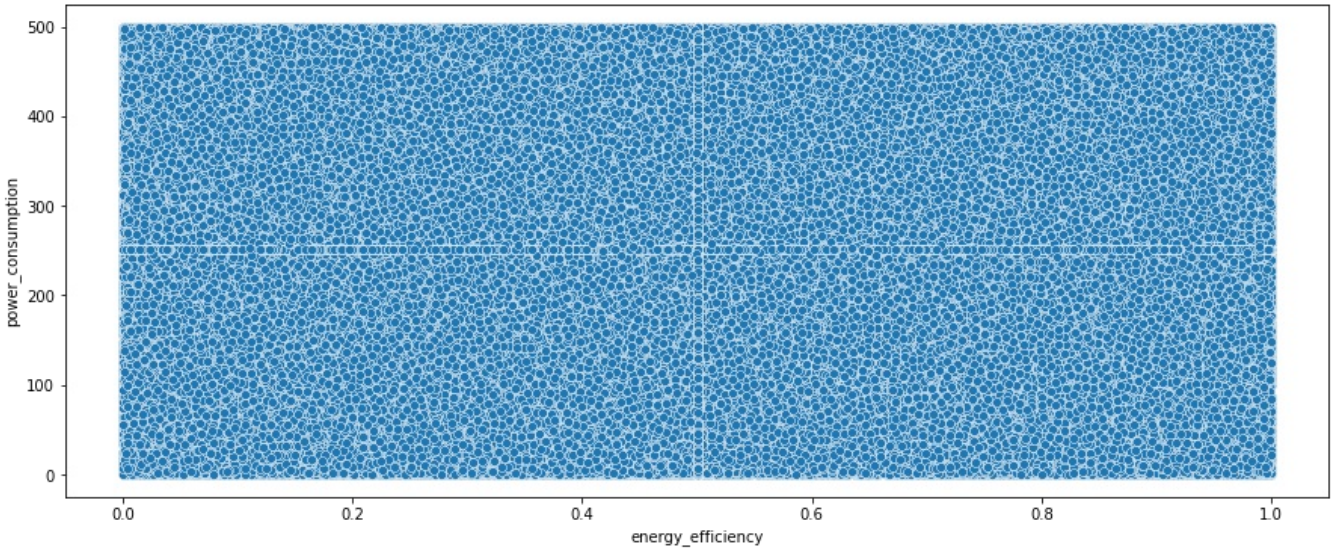

power_consumption
cpu_usage
0
100
200
300
400
500
0
20
40
60
80
100

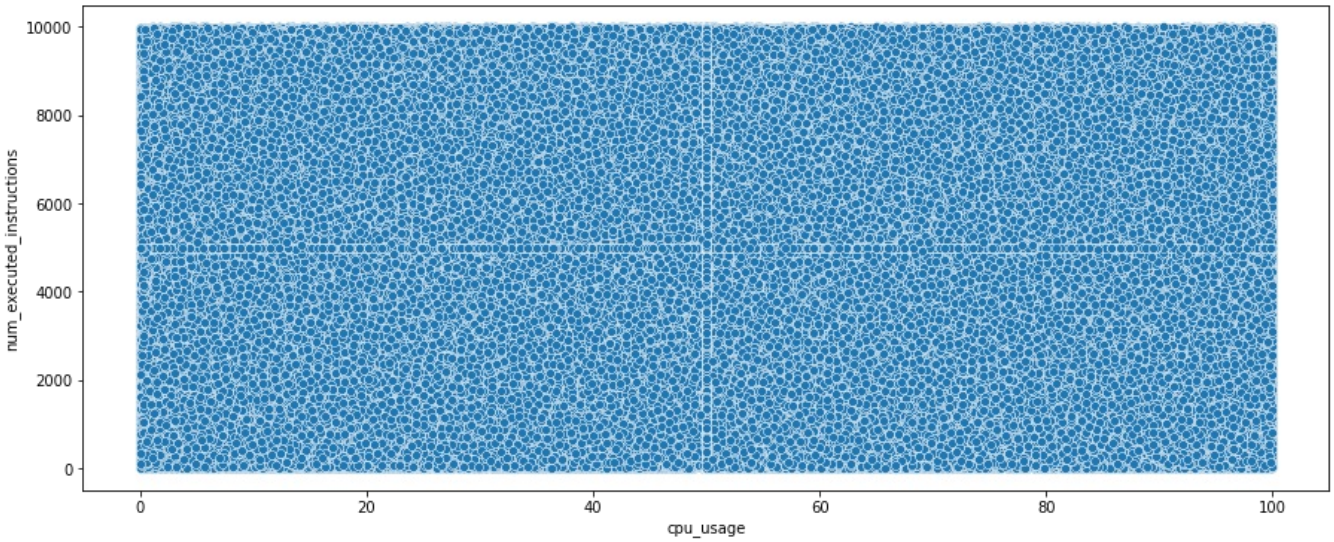

power_consumption
memory_usage
0
100
200
300
400
500
0
20
40
60
80
100

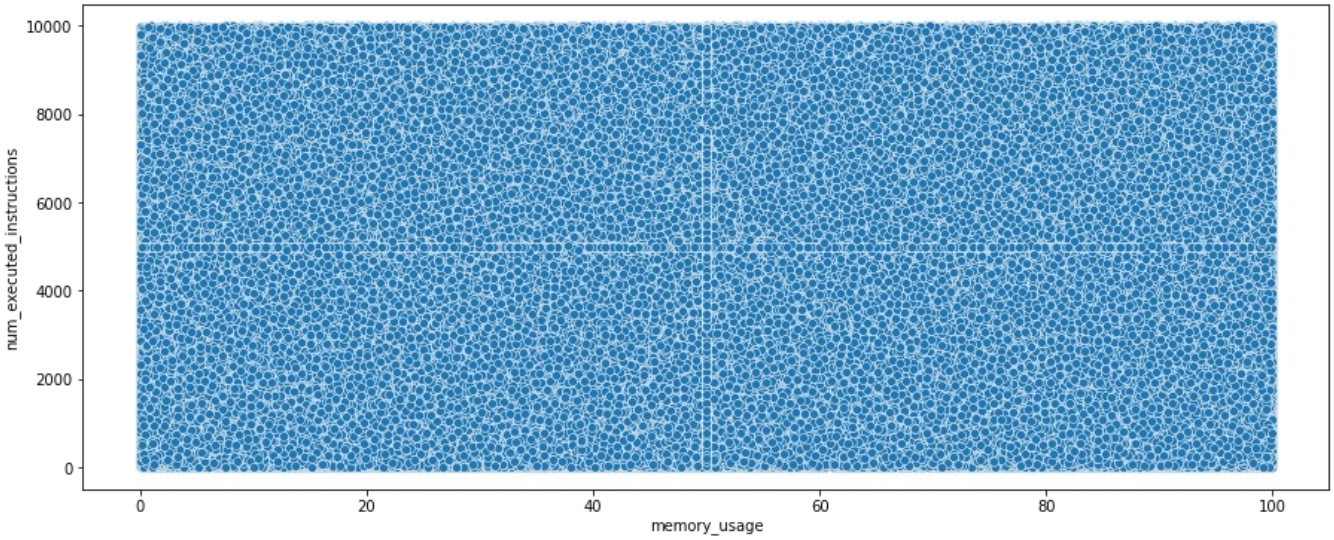

power_consumption
network_traffic
0
100
200
300
400
500
0
200
400
600
800
1000

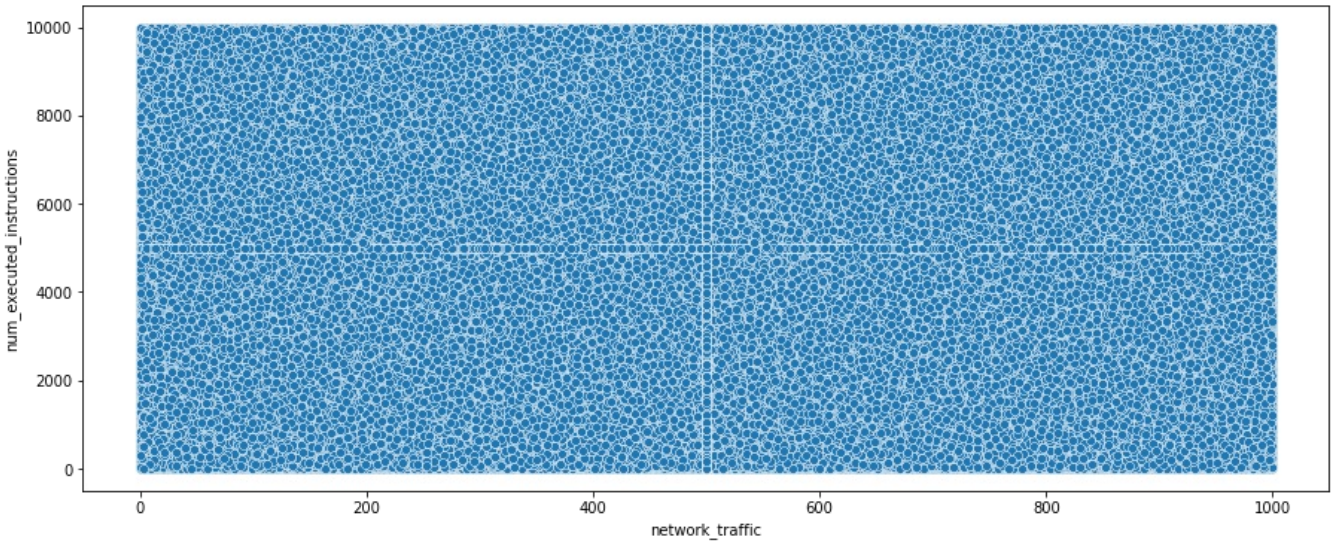

power_consumption
num_executed_instructions
0
100
200
300
400
500
0
2000
4000
6000
8000
10000

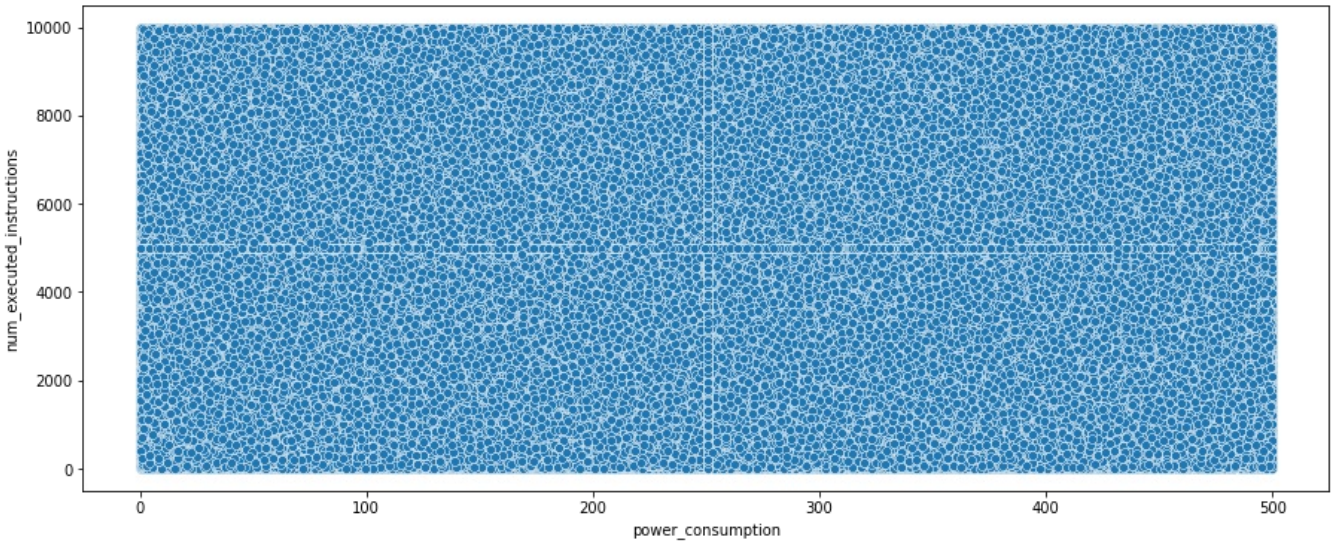

power_consumption
execution_time
0
100
200
300
400
500
0
20
40
60
80
100

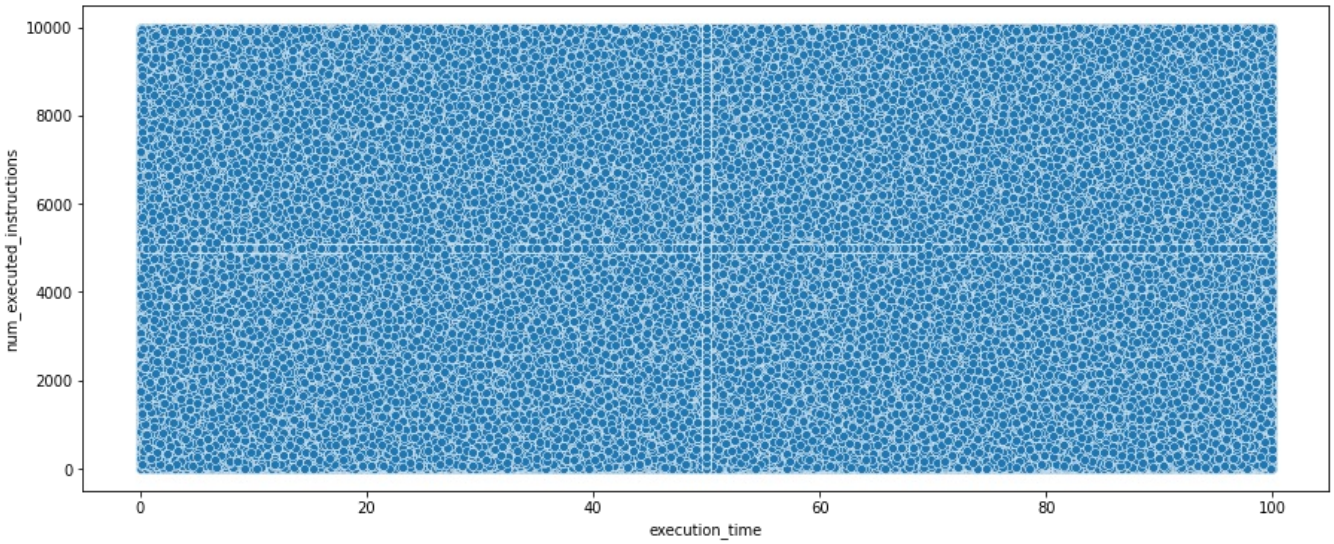

power_consumption
energy_efficiency
0
100
200
300
400
500
0.0
0.2
0.4
0.6
0.8
1.0

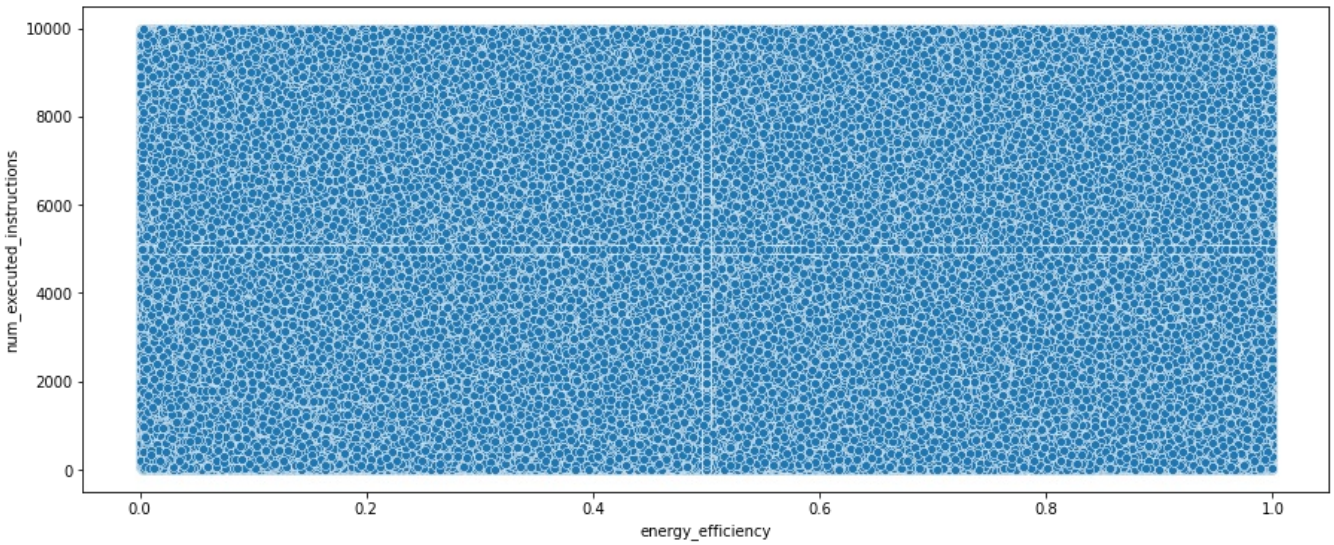

num_executed_instructions
10000
8000
6000
4000
2000
0
0
20
40
60
80
100
cpu_usage

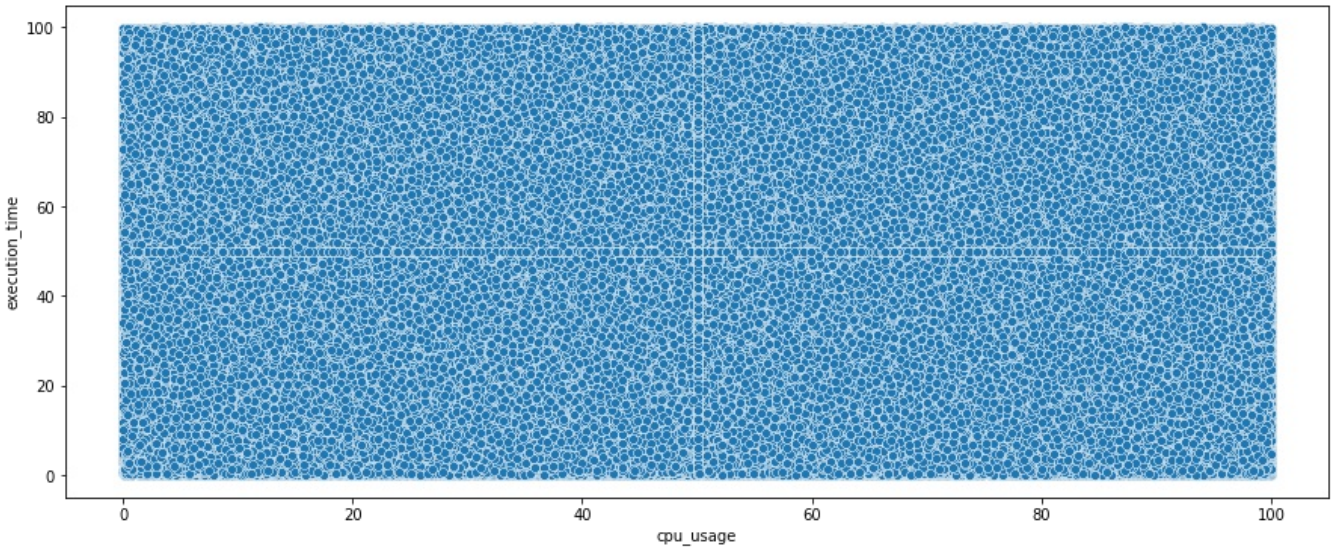

num_executed_instructions
10000
8000
6000
4000
2000
0
0
20
40
60
80
100
memory_usage

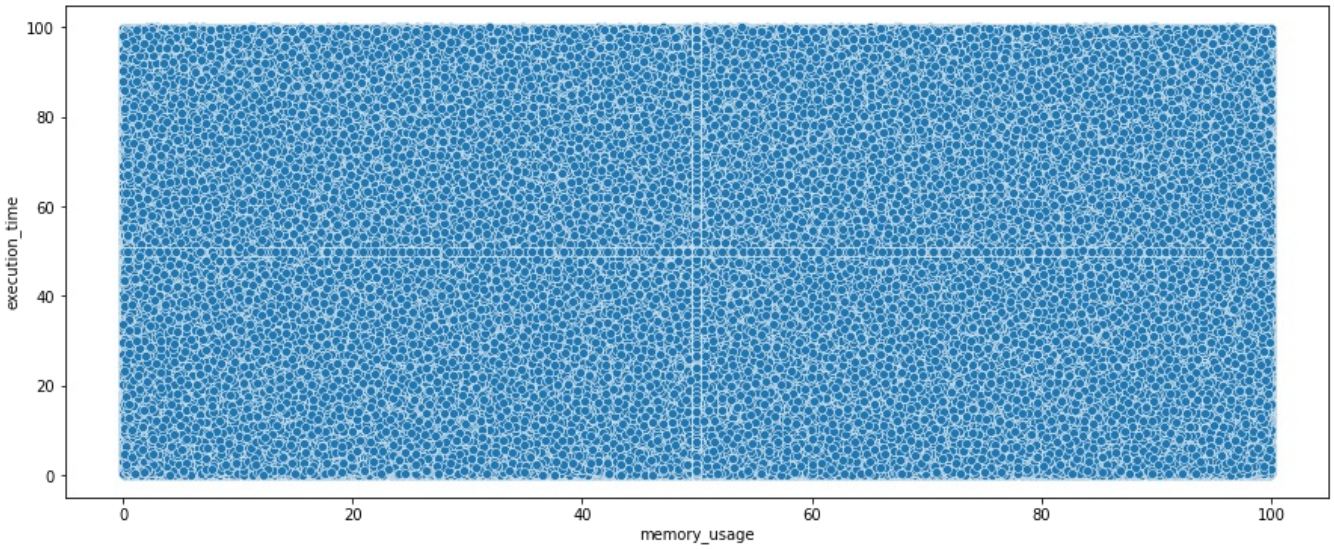

num_executed_instructions
10000
8000
6000
4000
2000
0
0
200
400
600
800
1000
network_traffic

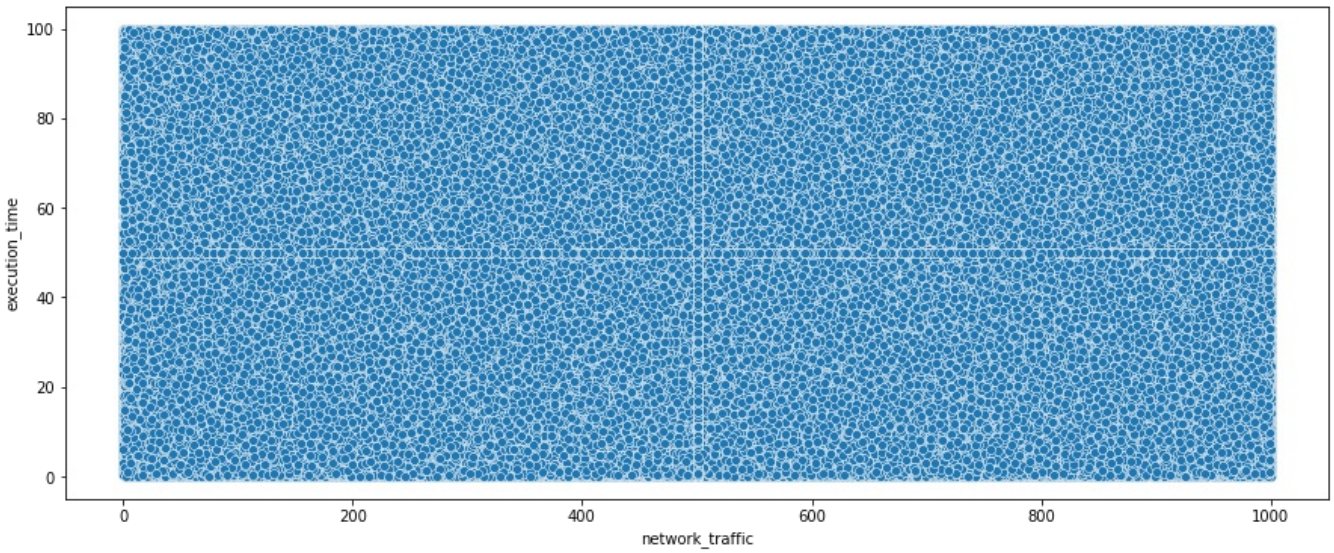

num_executed_instructions
power_consumption
0
2000
4000
6000
8000
10000
0
100
200
300
400
500

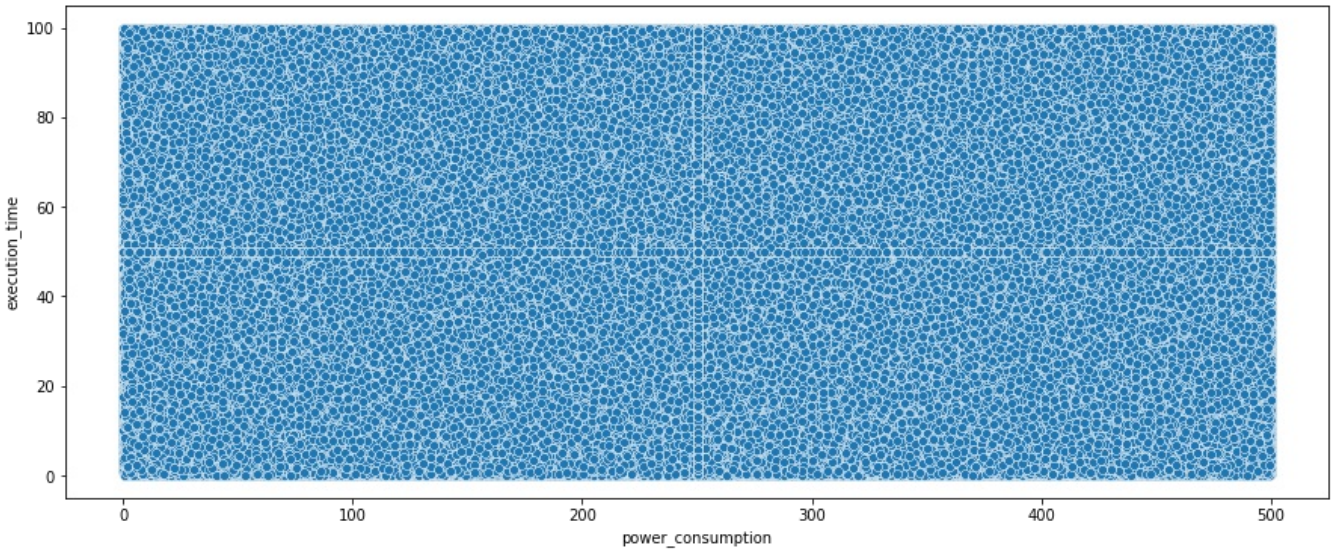

num_executed_instructions
execution_time
0
2000
4000
6000
8000
10000
0
20
40
60
80
100

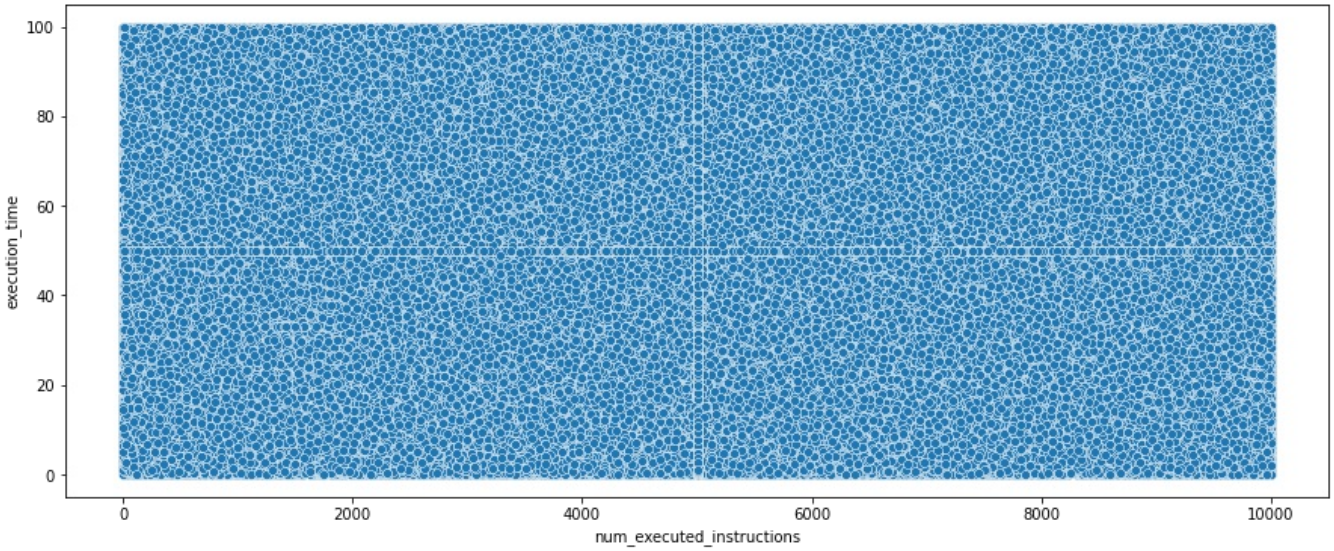

num_executed_instructions
energy_efficiency
0
2000
4000
6000
8000
10000
0.0
0.2
0.4
0.6
0.8
1.0

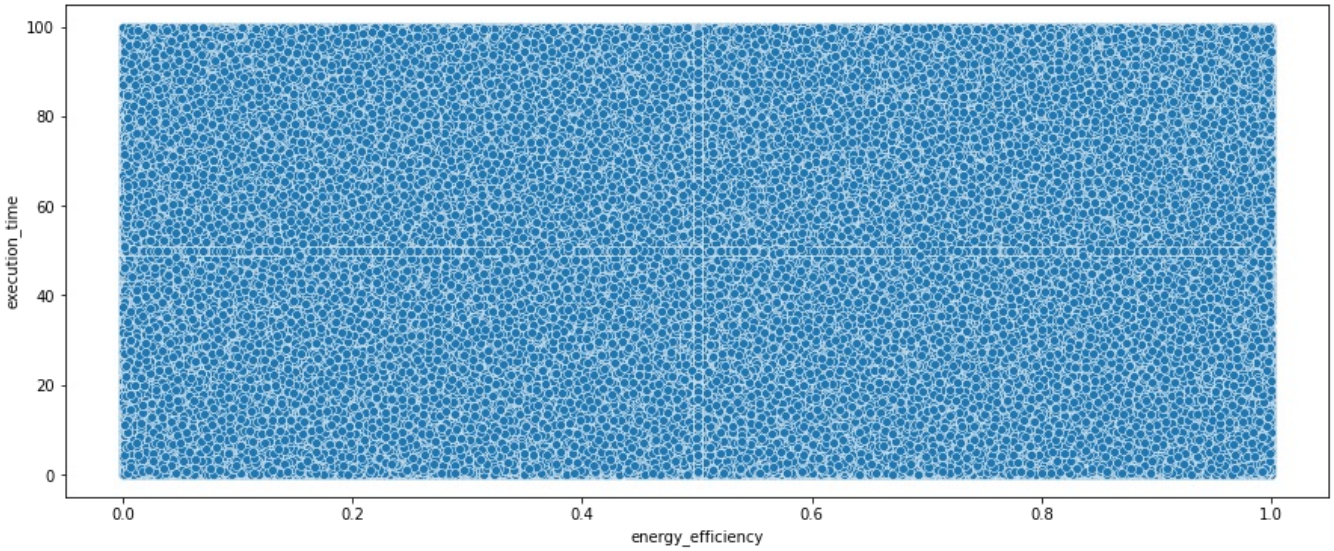

execution_time
cpu_usage
0
20
40
60
80
100

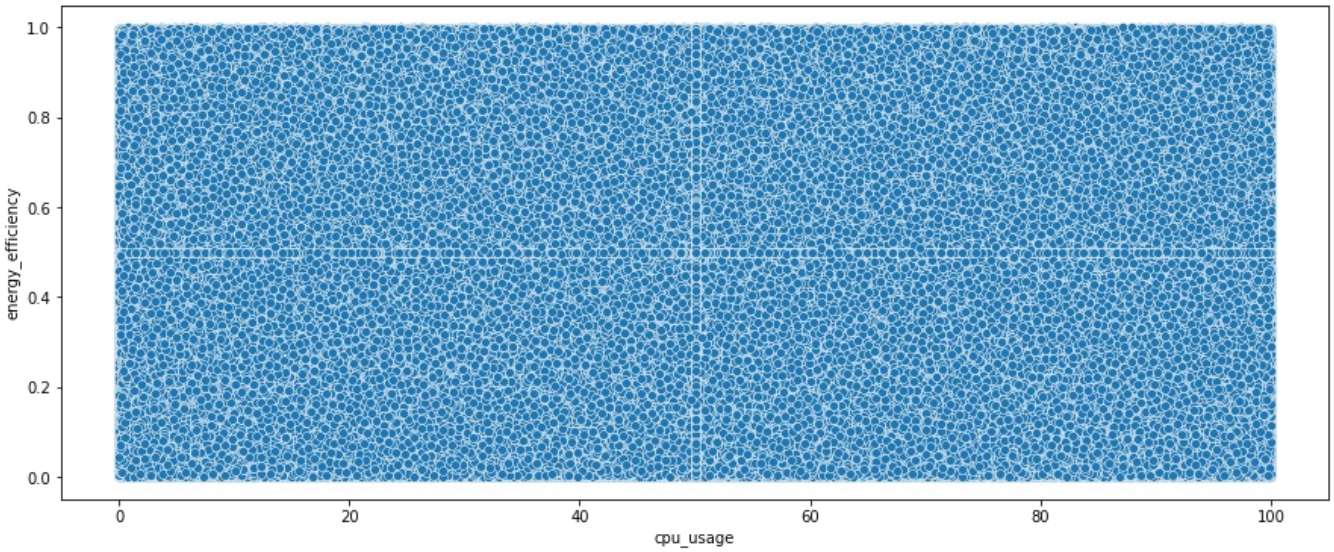

execution_time
memory_usage
0
20
40
60
80
100

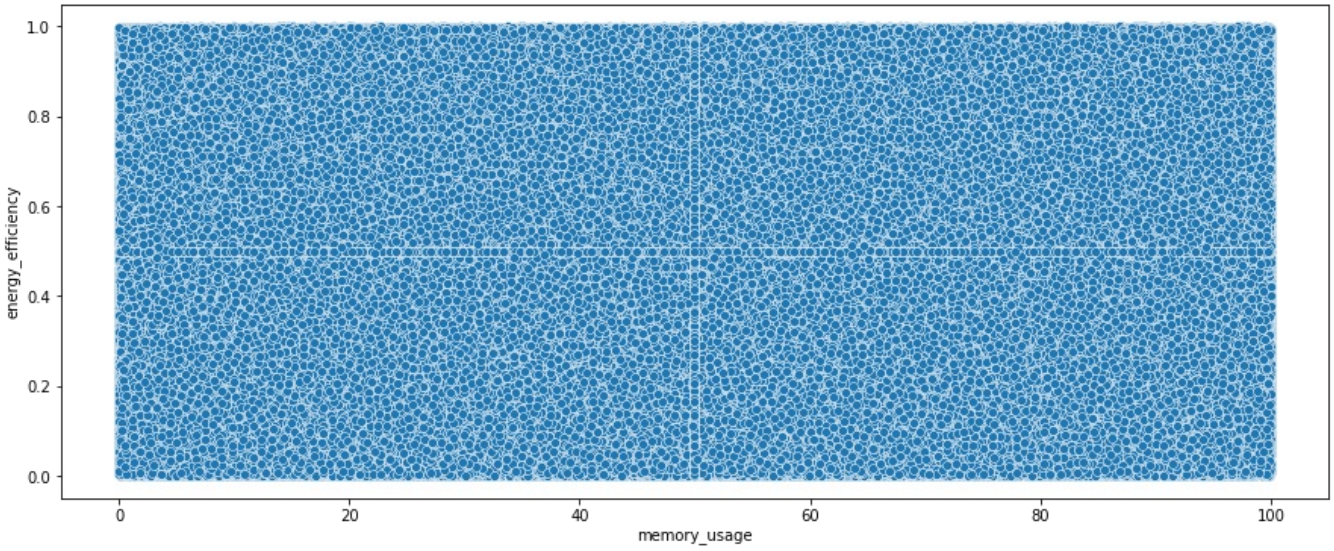

execution_time
network_traffic
0
200
400
600
800
1000

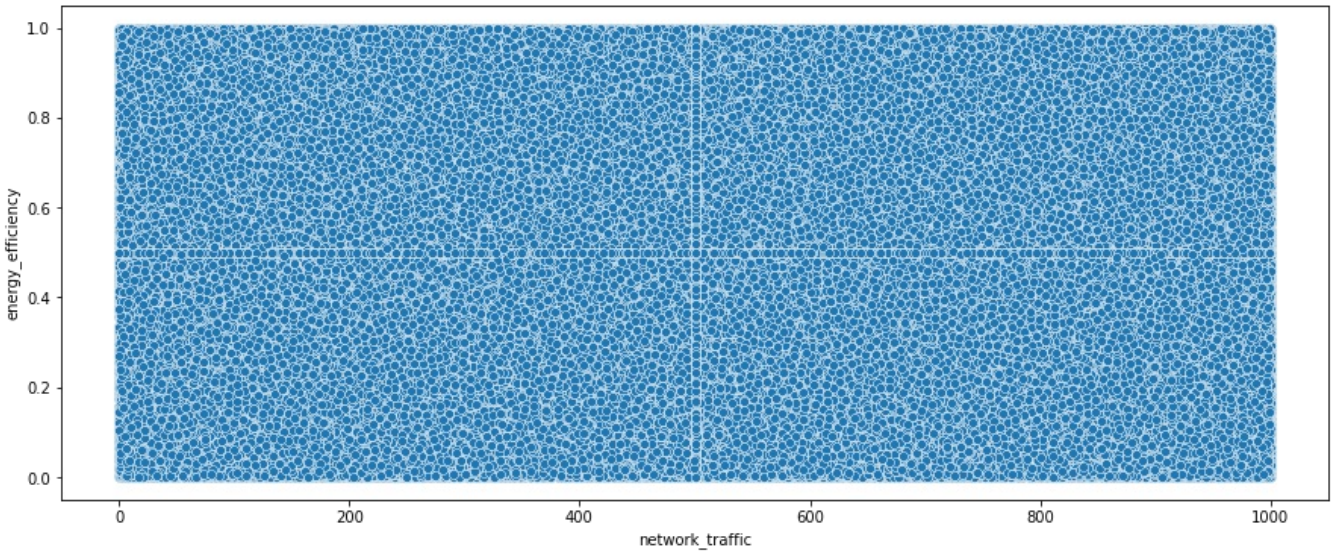

execution_time
power_consumption
0
20
40
60
80
100
0
100
200
300
400
500

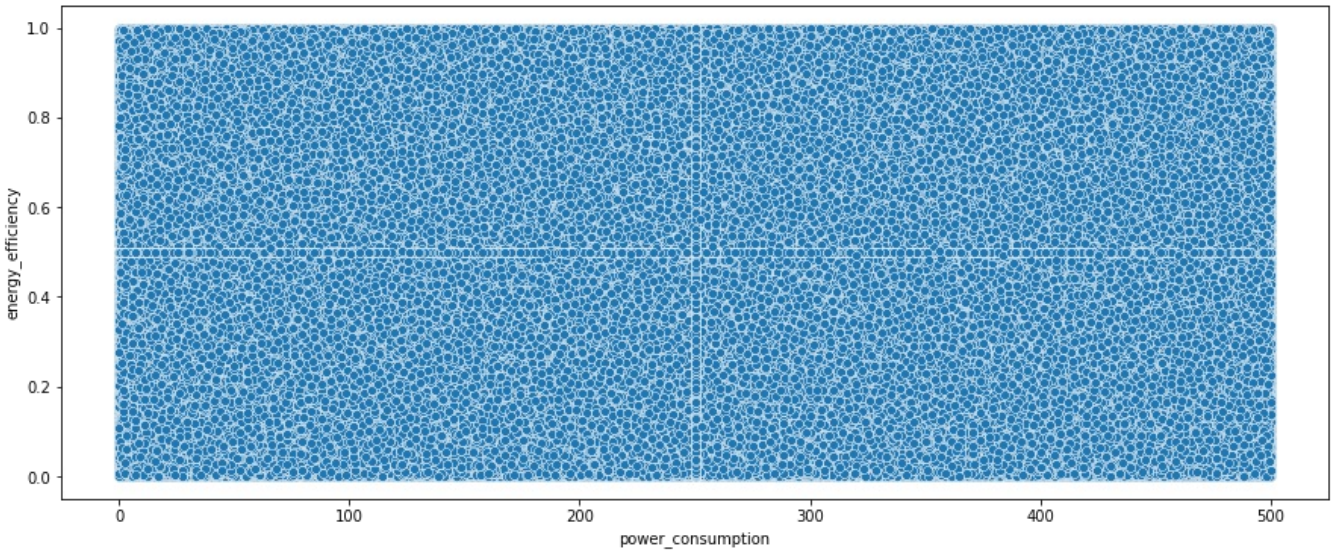

execution_time
num_executed_instructions
0
20
40
60
80
100
0
2000
4000
6000
8000
10000

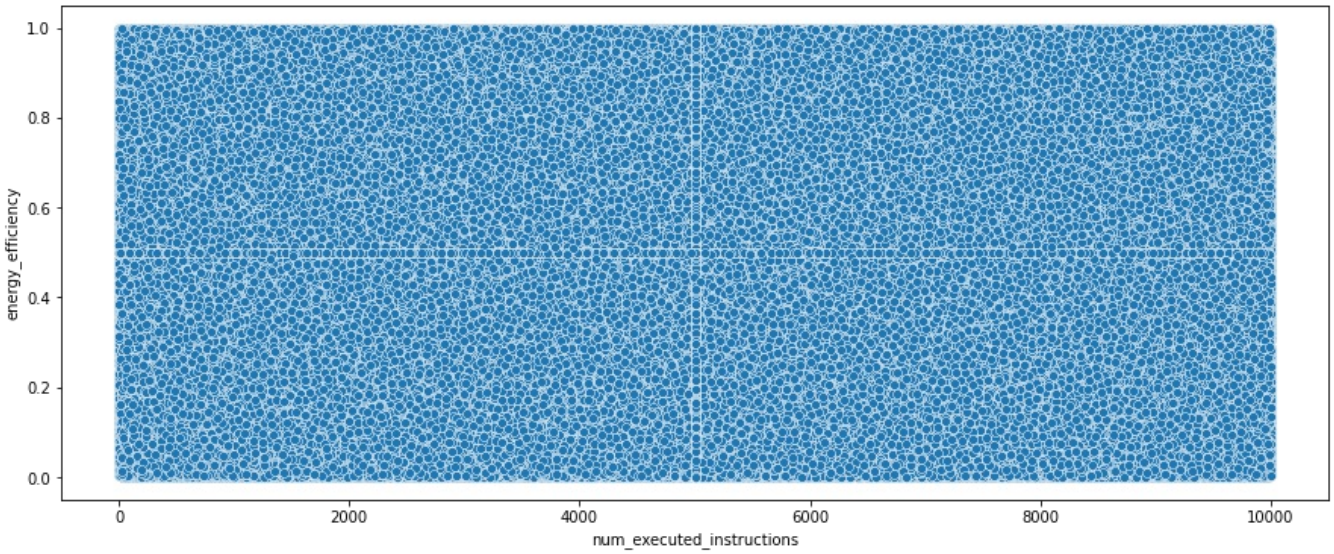

execution_time
energy_efficiency
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

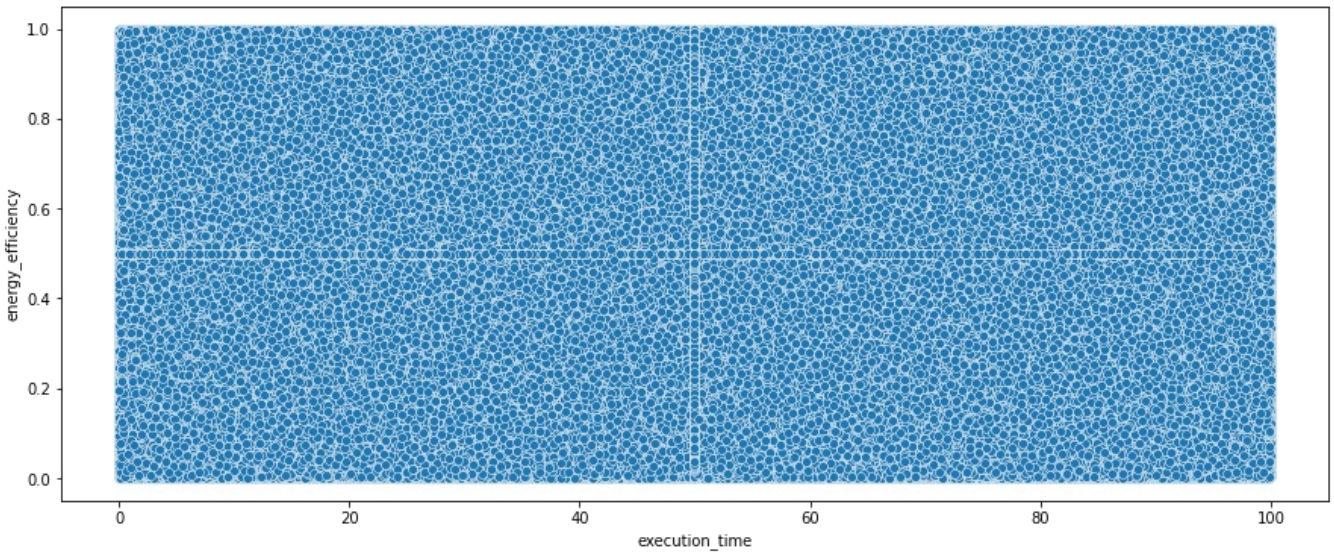

energy_efficiency
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
cpu_usage


energy_efficiency
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
memory_usage

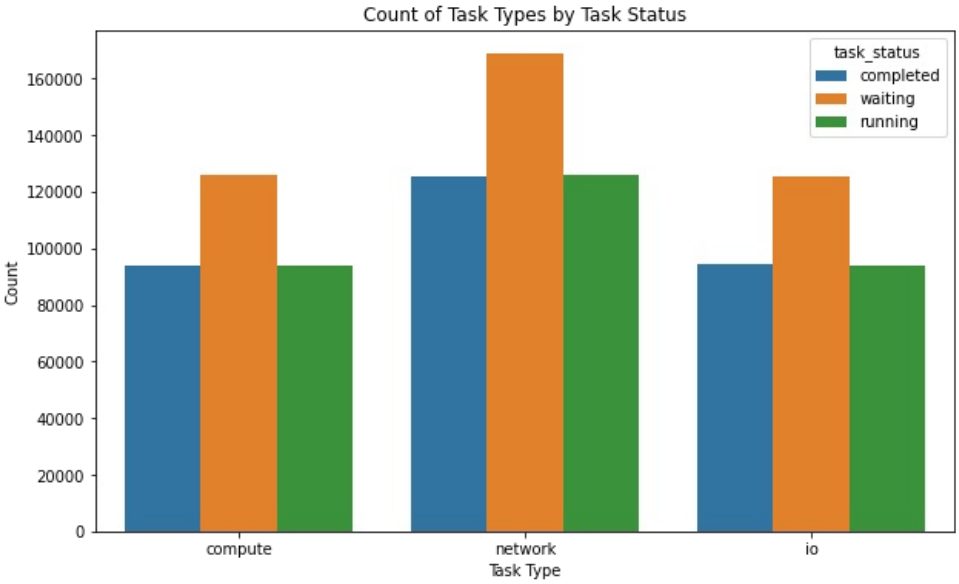

energy_efficiency
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
network_traffic



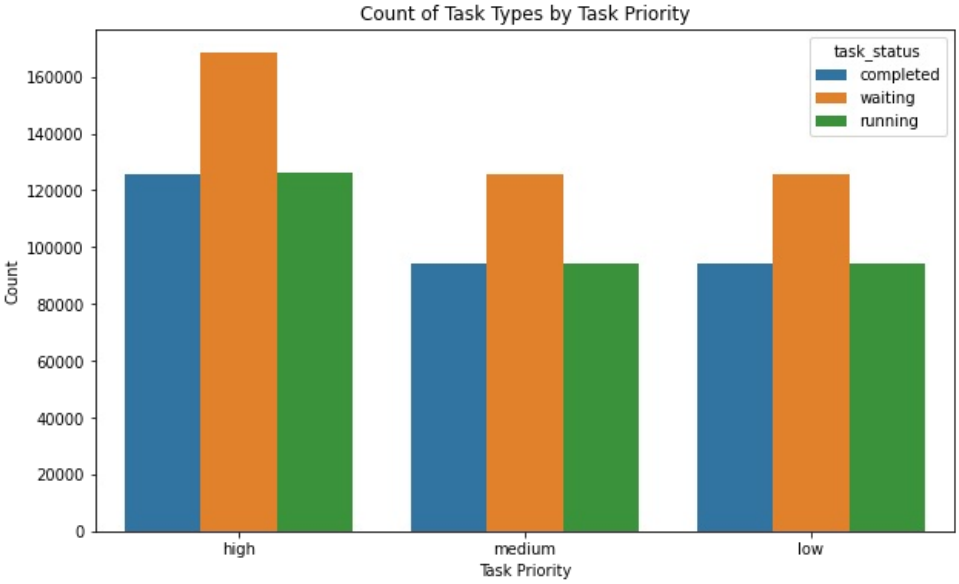




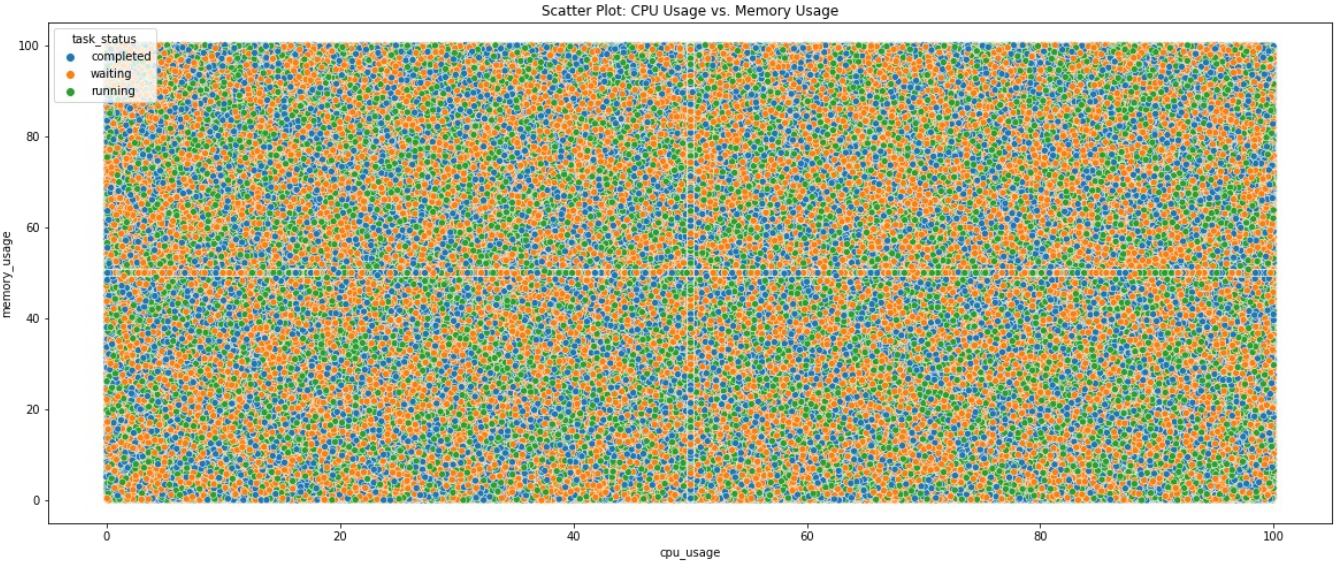
In [44]:
```
plt.figure(figsize=(10, 6))
sns.countplot(x='task_type', hue='task_status', data=df)
plt.title('Count of Task Types by Task Status')
plt.xlabel('Task Type')
plt.ylabel('Count')
plt.show()
```



In [45]:
```
plt.figure(figsize=(10, 6))
sns.countplot(x='task_priority', hue='task_status', data=df)
plt.title('Count of Task Types by Task Priority')
plt.xlabel('Task Priority')
plt.ylabel('Count')
plt.show()
```

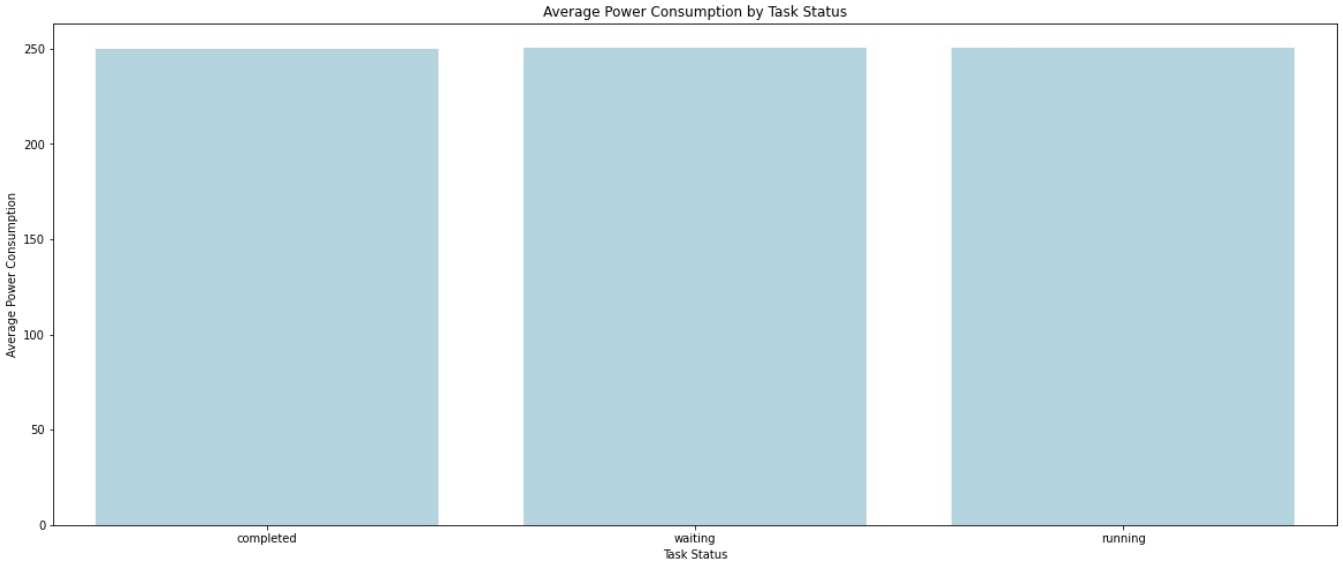


In [46]:
```
plt.figure(figsize=(20, 8))
sns.scatterplot(data=df, x='cpu_usage', y='memory_usage', hue='task_status')
plt.title('Scatter Plot: CPU Usage vs. Memory Usage')
plt.show()
```



In [47]:
```
plt.figure(figsize=(20, 8))
sns.barplot(data=df, x='task_status', y='power_consumption', ci=None, color='lightblue')
```

```
plt.title('Average Power Consumption by Task Status')
plt.xlabel('Task Status')
plt.ylabel('Average Power Consumption')
plt.show()
```



In [48]:
```
df = pd.get_dummies(df, columns=['task_type', 'task_priority', 'task_status'], prefix=['type', 'priority', 'tas
```

Fetaure Engineering

In [49]:
```
df['hour_of_day'] = df['timestamp'].dt.hour
df['day_of_week'] = df['timestamp'].dt.dayofweek
df['month'] = df['timestamp'].dt.month
```

In [50]:
```
daily_avg_cpu = df.groupby(pd.Grouper(key='timestamp', freq='D'))['cpu_usage'].mean()
```

In [51]:
```
daily_avg_cpu
```

Out[51]:
```
timestamp
2023-01-01    49.967117
2023-01-02    50.696723
2023-01-03    49.877983
2023-01-04    49.768051
2023-01-05    49.584332
                ...
2023-12-03    50.147385
2023-12-04    49.569591
2023-12-05    50.718439
2023-12-06    49.944509
2023-12-07    49.816105
Freq: D, Name: cpu_usage, Length: 341, dtype: float64
```

In [52]:
```
df['cpu_memory_interaction'] = df['cpu_usage'] * df['memory_usage']
```

In [53]:
```
df['network_power_ratio'] = df['network_traffic'] / df['power_consumption']
```

In [54]:
```
df.columns
```

Out[54]:
```
Index(['vm_id', 'timestamp', 'cpu_usage', 'memory_usage', 'network_traffic',
       'power_consumption', 'num_executed_instructions', 'execution_time',
       'energy_efficiency', 'type_compute', 'type_io', 'type_network',
       'priority_high', 'priority_low', 'priority_medium', 'task_completed',
       'task_running', 'task_waiting', 'hour_of_day', 'day_of_week', 'month',
       'cpu_memory_interaction', 'network_power_ratio'],
      dtype='object')
```

In [55]:
```
df
```

Out[55]:

	vm_id	timestamp	cpu_usage	memory_usage	network_traffic	power_consumption	num_executed_instructions	execution_tim
454772	17fa09d5-4039-4adf-b8c2-92eb116d940a	2023-01-01 00:00:00	50.048163	90.540824	845.326434	35.834837	6556.0	33.23828
898414	fe5e04e3-4ea4-48cf-a193-3f58abd147ed	2023-01-01 00:00:00	14.019569	74.367251	600.520958	368.735533	8240.0	50.74239
969845	2206b2b6-a41c-4755-93d9-2a1784ba66a9	2023-01-01 00:00:00	29.608038	58.788456	345.979118	202.838902	8774.0	40.73441
969844	NaN	2023-01-01 00:00:00	30.719806	89.199261	604.263594	325.820755	5002.0	3.23236
454771	a5ee77f0-5a60-474d-8a66-11bf38337326	2023-01-01 00:00:00	63.282078	49.963042	212.708467	358.127312	3983.0	45.30299
...	...	...	...	...	...	...	...	.
536024	2983b165-6811-4d50-bf63-f2e0bebd893e	2023-12-07 23:59:00	80.090971	47.878890	710.638233	183.238495	3821.0	11.91968
579029	a8120feb-7766-4356-b50c-22c67dd39deb	2023-12-07 23:59:00	30.351574	75.188533	70.264991	249.273443	5002.0	86.87770
393031	f9aa933d-e9a2-472a-8d2b-2aa3e85a22f0	2023-12-07 23:59:00	58.879801	91.365817	47.137432	156.109213	5473.0	57.10734
393030	9b4430c3-15cf-4a36-92c9-bc247b4efa21	2023-12-07 23:59:00	62.949990	43.526760	828.012861	190.164149	4490.0	74.07574
405795	10eac306-4e22-462c-af37-96a4789cc258	2023-12-07 23:59:00	95.447495	93.224728	266.471922	402.789075	6526.0	61.58673

1048575 rows × 23 columns

In [56]:
```
df = df.drop(['vm_id', 'timestamp'], axis = 1)
```

In [57]:
```
df.columns
```

Out[57]:
```
Index(['cpu_usage', 'memory_usage', 'network_traffic', 'power_consumption',
       'num_executed_instructions', 'execution_time', 'energy_efficiency',
       'type_compute', 'type_io', 'type_network', 'priority_high',
       'priority_low', 'priority_medium', 'task_completed', 'task_running',
       'task_waiting', 'hour_of_day', 'day_of_week', 'month',
       'cpu_memory_interaction', 'network_power_ratio'],
      dtype='object')
```

In [58]:
```
correlation_matrix = df.corr()
```

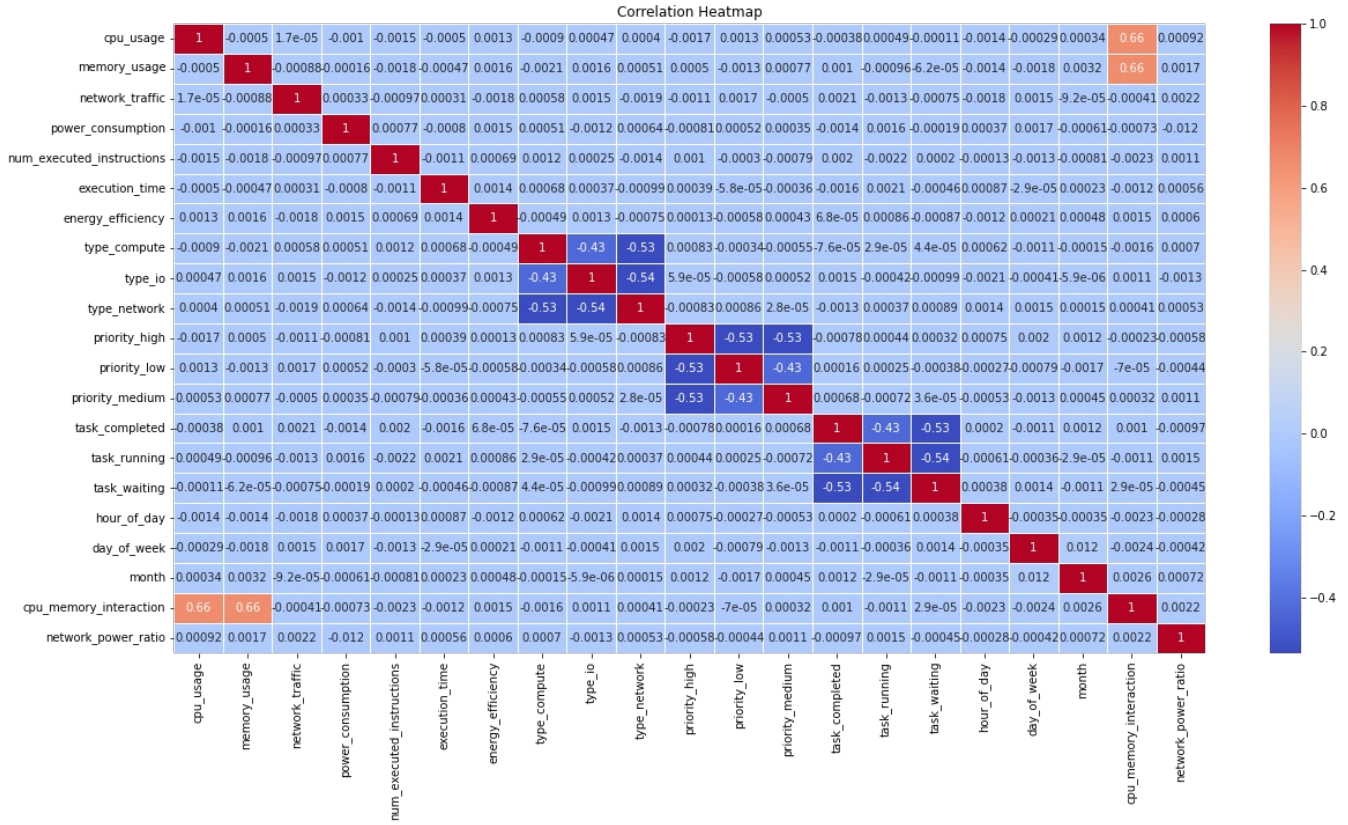
In [59]:
```
correlation_matrix
```

Out[59]:

	cpu_usage	memory_usage	network_traffic	power_consumption	num_executed_instructions	execution_time	ener
cpu_usage	1.000000	-0.000502	0.000017	-0.001031	-0.001528	-0.000498	
memory_usage	-0.000502	1.000000	-0.000882	-0.000161	-0.001799	-0.000471	
network_traffic	0.000017	-0.000882	1.000000	0.000326	-0.000969	0.000313	
power_consumption	-0.001031	-0.000161	0.000326	1.000000	0.000770	-0.000798	
num_executed_instructions	-0.001528	-0.001799	-0.000969	0.000770	1.000000	-0.001115	
execution_time	-0.000498	-0.000471	0.000313	-0.000798	-0.001115	1.000000	
energy_efficiency	0.001308	0.001562	-0.001797	0.001544	0.000688	0.001409	
type_compute	-0.000899	-0.002139	0.000581	0.000509	0.001221	0.000683	
type_io	0.000473	0.001594	0.001497	-0.001197	0.000248	0.000375	
type_network	0.000398	0.000508	-0.001942	0.000643	-0.001372	-0.000989	
priority_high	-0.001675	0.000505	-0.001133	-0.000813	0.001024	0.000390	
priority_low	0.001257	-0.001308	0.001714	0.000520	-0.000302	-0.000058	
priority_medium	0.000535	0.000769	-0.000503	0.000350	-0.000793	-0.000359	
task_completed	-0.000381	0.001027	0.002131	-0.001392	0.001962	-0.001589	
task_running	0.000494	-0.000961	-0.001328	0.001593	-0.002177	0.002084	
task_waiting	-0.000106	-0.000062	-0.000751	-0.000189	0.000201	-0.000464	
hour_of_day	-0.001415	-0.001435	-0.001791	0.000371	-0.000133	0.000875	
day_of_week	-0.000286	-0.001849	0.001505	0.001676	-0.001267	-0.000029	
month	0.000338	0.003202	-0.000092	-0.000609	-0.000810	0.000231	
cpu_memory_interaction	0.659280	0.659122	-0.000410	-0.000728	-0.002304	-0.001218	
network_power_ratio	0.000921	0.001712	0.002195	-0.011602	0.001124	0.000562	

21 rows × 21 columns

In [60]:
```
plt.figure(figsize=(20, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()
```



In [61]:
```
network_traffic_correlations = correlation_matrix['network_traffic'].sort_values(ascending=False)
```

In [62]:
```
network_traffic_correlations
```

Out[62]:
```
network_traffic              1.000000
network_power_ratio          0.002195
task_completed               0.002131
priority_low                 0.001714
day_of_week                  0.001505
type_io                      0.001497
type_compute                 0.000581
power_consumption            0.000326
execution_time               0.000313
cpu_usage                    0.000017
month                       -0.000092
cpu_memory_interaction      -0.000410
priority_medium             -0.000503
task_waiting                -0.000751
memory_usage                -0.000882
num_executed_instructions   -0.000969
priority_high               -0.001133
task_running                -0.001328
hour_of_day                 -0.001791
energy_efficiency           -0.001797
type_network                -0.001942
Name: network_traffic, dtype: float64
```

In [63]:
```
selected_columns = [
    'cpu_usage', 'memory_usage', 'network_traffic', 'power_consumption',
    'num_executed_instructions', 'execution_time', 'energy_efficiency',
    'type_compute', 'type_io', 'type_network', 'priority_high',
    'priority_low', 'priority_medium', 'task_completed', 'task_running',
    'task_waiting', 'hour_of_day', 'day_of_week', 'month',
    'cpu_memory_interaction', 'network_power_ratio'
]
```

In [64]:
```
correlation_df = df[selected_columns]
```

In [65]:
```
correlation_matrix = correlation_df.corr()
```

In [66]:
```
highly_correlated_pairs = []

for i in range(len(selected_columns)):
    for j in range(i+1, len(selected_columns)):
        feature1 = selected_columns[i]
        feature2 = selected_columns[j]
        correlation = correlation_matrix.loc[feature1, feature2]

        if abs(correlation) > 0.6:
            highly_correlated_pairs.append((feature1, feature2, correlation))

for pair in highly_correlated_pairs:
    feature1, feature2, correlation = pair
    print(f"Correlation between {feature1} and {feature2}: {correlation}")
```

```
Correlation between cpu_usage and cpu_memory_interaction: 0.6592797937904734
Correlation between memory_usage and cpu_memory_interaction: 0.6591217402304457
```

Data Modeling

In the dataset, which includes performance metrics in a cloud computing environment, several features could be related to privacy and security in cloud computing:

1. **Network Traffic:** Network traffic can be a crucial feature in assessing security. Anomalies or unusual spikes in network traffic may indicate a security breach or a potential privacy threat.
2. **Power Consumption:** Monitoring power consumption is important for resource management in cloud computing, but it can also indirectly relate to security. Unusual power consumption patterns could be a sign of unauthorized access or resource misuse.
3. **Task Type and Task Priority:** The type and priority of tasks being executed in the cloud can impact security. High-priority tasks may be more critical and require stricter security measures.
4. **Task Status:** The task status can provide insights into the overall health of the cloud environment. Unusual task statuses could indicate security incidents.
5. **Execution Time and Number of Executed Instructions:** These metrics can indirectly relate to security. Long execution times or excessive instructions executed for a task could be signs of inefficient or malicious behavior.
6. **Energy Efficiency:** While primarily an operational metric, energy efficiency can indirectly relate to security. Inefficient resource usage may lead to security vulnerabilities.

It's important to note that the direct relationship between these features and security/privacy would require domain-specific analysis and context. These features alone may not provide a complete picture of security and privacy in a cloud computing environment. Additional security-specific features and context are often necessary for a comprehensive assessment of security and privacy in cloud computing.

In [67]:
```
df.columns
```

Out[67]:
```
Index(['cpu_usage', 'memory_usage', 'network_traffic', 'power_consumption',
       'num_executed_instructions', 'execution_time', 'energy_efficiency',
       'type_compute', 'type_io', 'type_network', 'priority_high',
       'priority_low', 'priority_medium', 'task_completed', 'task_running',
       'task_waiting', 'hour_of_day', 'day_of_week', 'month',
       'cpu_memory_interaction', 'network_power_ratio'],
      dtype='object')
```

In [68]:
```
df1 = df.copy()
```

MinMaxScaler is a feature scaling technique commonly used in machine learning to transform numerical data into a specific range, typically between 0 and 1. It's a type of data normalization or feature scaling that helps make different features or variables more comparable and can be particularly useful for algorithms that are sensitive to the scale of input features, such as gradient-based optimization methods.

Here's how MinMaxScaler works:

Scaling to a Specified Range: MinMaxScaler scales your data to a specified range, often between 0 and 1. You can also specify a different range if needed.

Preserving Relative Relationships: It preserves the relative relationships between the original values. That means if you have data with varying scales, Min-Max scaling will ensure that these relationships are maintained.

Transforming Data: The transformation is done by applying a linear transformation to each feature independently. The formula for Min-Max scaling is as follows:

X_scaled = (X - X_min) / (X_max - X_min) Where:

X is the original value of the feature. X_min is the minimum value of the feature in the dataset. X_max is the maximum value of the feature in the dataset. Normalization: The scaled feature values will be within the specified range, and the minimum value will be 0, while the maximum value will be 1.

Here's an example of how to use MinMaxScaler in Python with scikit-learn:

from sklearn.preprocessing import MinMaxScaler

Create a MinMaxScaler instance

scaler = MinMaxScaler()

Fit and transform the data

scaled_data = scaler.fit_transform(original_data) Common use cases for MinMaxScaler include when you want to scale features that have different units or ranges, or when you want to ensure that all features have values within a consistent range, which can help some machine learning algorithms converge faster and perform better.

In [69]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
features_to_scale = ['cpu_usage', 'memory_usage', 'network_traffic', 'power_consumption',
                     'num_executed_instructions', 'execution_time', 'energy_efficiency', 'network_power_ratio',
                    'cpu_memory_interaction']

df1[features_to_scale] = scaler.fit_transform(df1[features_to_scale])
```

In [70]:
```
df1
```

Out[70]:

	cpu_usage	memory_usage	network_traffic	power_consumption	num_executed_instructions	execution_time	energy_efficiency	type_
454772	0.500481	0.905409	0.845327	0.071669	0.655666	0.332383	0.451265	
898414	0.140195	0.743673	0.600521	0.737472	0.824082	0.507424	0.639540	
969845	0.296080	0.587885	0.345979	0.405678	0.877488	0.407344	0.890145	
969844	0.307198	0.891993	0.604264	0.651642	0.500250	0.032324	0.992170	
454771	0.632821	0.499631	0.212708	0.716255	0.398340	0.453030	0.681051	
...	...	...	...	...	...	...	...	
536024	0.800910	0.478789	0.710638	0.366477	0.382138	0.119197	0.793805	
579029	0.303515	0.751886	0.070265	0.498547	0.500250	0.868777	0.418236	
393031	0.588798	0.913659	0.047137	0.312219	0.547355	0.571074	0.335630	
393030	0.629500	0.435268	0.828013	0.380329	0.449045	0.740758	0.375720	
405795	0.954475	0.932248	0.266472	0.805579	0.652665	0.615868	0.396133	

1048575 rows × 21 columns

In [71]:
```
from sklearn.model_selection import train_test_split
```

In [72]:
```
X = df.drop(['network_traffic', 'cpu_memory_interaction'], axis=1)
y = df['network_traffic']
```

In [73]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

In [74]:
```
from sklearn.metrics import mean_squared_error, r2_score
```

In [75]:
```
from sklearn.linear_model import LinearRegression
```

In [76]:
```
lr_regressor = LinearRegression()
lr_regressor.fit(X_train, y_train)
```

Out[76]:
```
▾ LinearRegression
LinearRegression()
```

In [77]:
```
y_pred = lr_regressor.predict(X_test)
```

In [78]:
```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared:", r2)
```

```
Mean Squared Error: 75014.80092513883
R-squared: -1.3288719485915479e-05
```

PolynomialFeatures is a preprocessing technique commonly used in machine learning and feature engineering, particularly in the context of polynomial regression and other polynomial models. It's a transformation method provided by libraries like Scikit-Learn in Python. The main purpose of PolynomialFeatures is to generate polynomial and interaction features from the original features in your dataset.

Here's an explanation of how PolynomialFeatures works:

Polynomial Expansion: It takes a set of original features and expands them to include polynomial features of a specified degree. For example, if you have a single feature 'x' and you set the degree to 2, it will create new features like 'x^2', 'x^3', and so on. This is useful when the relationship between the features and the target variable is nonlinear, and a linear model cannot capture it adequately.

Interaction Features: In addition to generating polynomial features, it can also create interaction features. Interaction features are the products of pairs of features. For example, if you have features 'x' and 'y', it can create 'x*y' as a new feature. This allows the model to account for interactions between different features.

Combinatorial Explosion: Be cautious when using high degrees or many interaction terms, as it can lead to a combinatorial explosion of features, increasing the complexity of the model and potentially causing overfitting.

Regularization: The use of polynomial and interaction features can sometimes lead to overfitting, especially if the degree is very high. To mitigate this, regularization techniques like Lasso or Ridge regression can be used.

In [79]:
```
from sklearn.preprocessing import PolynomialFeatures
```

In [80]:
```
poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)
```

In [81]:
```
regressor = LinearRegression()
```

```
regressor.fit(X_train_poly, y_train)
```

Out[81]:
```
▾ LinearRegression
LinearRegression()
```

In [82]:
```
y_pred = regressor.predict(X_test_poly)
```

In [83]:
```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared:", r2)
```

```
Mean Squared Error: 2.4783057739602263e-16
R-squared: 1.0
```

In [84]:
```
from sklearn.linear_model import Lasso
```

In [85]:
```
lasso = Lasso(alpha=0.01)
lasso.fit(X_train_poly, y_train)
```

Out[85]:
```
▾ Lasso
Lasso(alpha=0.01)
```

In [86]:
```
y_pred_lasso = lasso.predict(X_test_poly)
```

In [87]:
```
mse_lasso = mean_squared_error(y_test, y_pred_lasso)
r2_lasso = r2_score(y_test, y_pred_lasso)

print("Lasso Regression:")
print("Mean Squared Error:", mse_lasso)
print("R-squared:", r2_lasso)
```

```
Lasso Regression:
Mean Squared Error: 0.13458457578218824
R-squared: 0.9999982058692074
```

In [88]:
```
from sklearn.linear_model import Ridge
```

In [89]:
```
ridge = Ridge(alpha=1.0)
ridge.fit(X_train_poly, y_train)
```

Out[89]:
```
▾ Ridge
Ridge()
```

In [90]:
```
y_pred_ridge = ridge.predict(X_test_poly)
```

In [91]:
```
mse_ridge = mean_squared_error(y_test, y_pred_ridge)
r2_ridge = r2_score(y_test, y_pred_ridge)

print("Ridge Regression:")
print("Mean Squared Error:", mse_ridge)
print("R-squared:", r2_ridge)
```

```
Ridge Regression:
Mean Squared Error: 2.931350535669207e-17
R-squared: 1.0
```

In [92]:
```
from sklearn.tree import DecisionTreeRegressor
```

In [93]:
```
dt_regressor = DecisionTreeRegressor()
dt_regressor.fit(X_train_poly, y_train)
```

Out[93]:
```
▾ DecisionTreeRegressor
DecisionTreeRegressor()
```

In [94]:
```
y_pred_dt = dt_regressor.predict(X_test_poly)
```

In [95]:
```
mse_dt = mean_squared_error(y_test, y_pred_dt)
r2_dt = r2_score(y_test, y_pred_dt)

print("Decision Tree Regressor:")
print("Mean Squared Error:", mse_dt)
print("R-squared:", r2_dt)
```

```
Decision Tree Regressor:
Mean Squared Error: 1.6192734444512145e-05
R-squared: 0.9999999997841366
```

In [96]:
```
from sklearn.ensemble import RandomForestRegressor
```

In [97]:
```
rf_regressor = RandomForestRegressor(n_estimators = 5, max_depth = 5, min_samples_split = 2,
                                     min_samples_leaf = 1)
rf_regressor.fit(X_train_poly, y_train)
```

Out[97]:
```
RandomForestRegressor
RandomForestRegressor(max_depth=5, n_estimators=5)
```

In [98]:
```
y_pred_rf = rf_regressor.predict(X_test_poly)
```

In [99]:
```
mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

print("Random Forest Regressor with Hyperparameters:")
print("Mean Squared Error:", mse_rf)
print("R-squared:", r2_rf)
```

```
Random Forest Regressor with Hyperparameters:
Mean Squared Error: 74.41891568619484
R-squared: 0.9990079303857307
```

In [100…
```
import xgboost as xgb
```

In [101…
```
xgb_regressor = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=5, max_depth=3, learning_rate=0.1)
xgb_regressor.fit(X_train_poly, y_train)
```

Out[101]:
```
XGBRegressor
XGBRegressor(base_score=0.5, booster='gbtree', callbacks=None,
             colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
             early_stopping_rounds=None, enable_categorical=False,
             eval_metric=None, gamma=0, gpu_id=-1, grow_policy='depthwise',
             importance_type=None, interaction_constraints='',
             learning_rate=0.1, max_bin=256, max_cat_to_onehot=4,
             max_delta_step=0, max_depth=3, max_leaves=0, min_child_weight=1,
             missing=nan, monotone_constraints='()', n_estimators=5, n_jobs=0,
             num_parallel_tree=1, predictor='auto', random_state=0, reg_alpha=0,
             reg_lambda=1, ...)
```

In [102…
```
y_pred_xgb = xgb_regressor.predict(X_test_poly)
```

In [103…
```
mse_xgb = mean_squared_error(y_test, y_pred_xgb)
r2_xgb = r2_score(y_test, y_pred_xgb)

print("XGBoost Regressor:")
print("Mean Squared Error:", mse_xgb)
print("R-squared:", r2_xgb)
```

```
XGBoost Regressor:
Mean Squared Error: 113815.20236492252
R-squared: -0.5172567735906362
```

In [104…
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

In [105…
```
from keras.models import Sequential
from keras.layers import Dense, Activation, BatchNormalization, Dropout
```

Batch Normalization is a technique used in deep learning to improve the training and performance of neural networks. It normalizes the input of each layer in a mini-batch of data. Here's what it does:

1. **Normalization**: For each mini-batch of data, Batch Normalization normalizes the activations by subtracting the mean and dividing by the standard deviation. This makes the distribution of the inputs more stable.

2. **Scaling and Shifting**: After normalization, Batch Normalization allows the model to learn two additional parameters (scaling and shifting) for each feature. This enables the network to adapt the normalization to the specific needs of each layer and each feature.

The key benefits of Batch Normalization include:

- **Stability**: It stabilizes and speeds up the training process. Normalizing activations prevents gradients from becoming too large, which can cause training to fail.

- **Regularization**: Batch Normalization acts as a form of regularization by adding noise to the activations. This can help prevent overfitting.

- **Faster Convergence**: Networks train faster and require fewer epochs to converge. The normalization helps to distribute the learning in a more uniform way across the network.

- **Improved Gradient Flow**: It mitigates vanishing and exploding gradient problems by normalizing the activations.
- **Network Architecture**: It can make it easier to train deep networks with many layers.

Batch Normalization is typically applied to the inputs of each layer and is used in various neural network architectures like convolutional neural networks (CNNs) and recurrent neural networks (RNNs). It's a powerful tool for improving the training and performance of deep learning models.

In [106…
```
model = Sequential()
```

Architecture is Subject to Changes for better score.

In [107…
```
model.add(Dense(units=32, input_shape=(X_train_poly.shape[1],)))
model.add(Activation('relu'))
model.add(BatchNormalization())

model.add(Dense(units=64))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(units=128))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(units=1))
model.add(Activation('linear'))
```

In [108…
```
model.compile(optimizer='adam', loss='mean_squared_error')
```

In [109…
```
history = model.fit(X_train_poly, y_train, epochs=10, batch_size=32, validation_data=(X_test_poly, y_test))
```

```
Epoch 1/10
22938/22938 [==============================] - 89s 4ms/step - loss: 56602.2188 - val_loss: 24633397248.0000
Epoch 2/10
22938/22938 [==============================] - 88s 4ms/step - loss: 19053.4414 - val_loss: 1323018354688.0000
Epoch 3/10
22938/22938 [==============================] - 71s 3ms/step - loss: 15566.4961 - val_loss: 78584040.0000
Epoch 4/10
22938/22938 [==============================] - 69s 3ms/step - loss: 11553.5098 - val_loss: 167153472.0000
Epoch 5/10
22938/22938 [==============================] - 71s 3ms/step - loss: 9631.6846 - val_loss: 4872241152.0000
Epoch 6/10
22938/22938 [==============================] - 67s 3ms/step - loss: 9269.3174 - val_loss: 3379766784.0000
Epoch 7/10
22938/22938 [==============================] - 81s 4ms/step - loss: 9120.7578 - val_loss: 92827.8672
Epoch 8/10
22938/22938 [==============================] - 87s 4ms/step - loss: 8876.7100 - val_loss: 503429.8750
Epoch 9/10
22938/22938 [==============================] - 83s 4ms/step - loss: 8846.2324 - val_loss: 101044144.0000
Epoch 10/10
22938/22938 [==============================] - 86s 4ms/step - loss: 8832.4443 - val_loss: 1816842624.0000
```

In [110…
```
y_pred_nn = model.predict(X_test_poly).flatten()
```

```
9831/9831 [==============================] - 13s 1ms/step
```

In [111…
```
mse_nn = mean_squared_error(y_test, y_pred_nn)
r2_nn = r2_score(y_test, y_pred_nn)

print("Deep Learning Regressor:")
print("Mean Squared Error:", mse_nn)
print("R-squared:", r2_nn)
```

```
Deep Learning Regressor:
Mean Squared Error: 1816911822.5330942
R-squared: -24220.032976916777
```