EDA

November 23, 2025

1 EDA: Google Cluster Trace 2019 - Demand-Only Analysis (CORRECTED)

This notebook performs **corrected** exploratory data analysis on workload demand from Google Cloud Traces 2019.

1.1 Critical Corrections Applied

1. **NO REAL TIMESTAMPS**: Google Trace 2019 does NOT contain real timestamps. All temporal analysis uses `bucket_index` (derived from `bucket_s`).
2. **Synthetic Time Features**: Hour-of-day, day-of-week features are synthetic and used ONLY for seasonality pattern detection.
3. **Machine Counts**: The `machines` column represents machines reporting usage, NOT total cluster capacity.
4. **Instance Events Mismatch**: `new_instances_cluster` includes tasks that may not appear in usage data - requires normalization.
5. **Memory Units**: Memory values are already normalized (likely in GB based on magnitude).

1.2 Data Available

- `data/processed/machine_level.parquet`: Per-machine CPU/memory demand
- `data/processed/cluster_level.parquet`: Aggregated cluster-wide demand

Columns: - `bucket_s`: Seconds since trace start (300s = 5min intervals) - `cpu_demand` / `cpu_used`: CPU usage in cores - `mem_demand` / `mem_used`: Memory usage (normalized, likely GB) - `machines`: Number of machines reporting usage in this bucket - `new_instances_cluster` / `new_instances_machine`: New instance arrivals

1.3 Analysis Structure

1. Setup & Data Loading
2. Data Quality & Coverage Analysis
3. Temporal Analysis with bucket_index
4. Demand Distribution & Statistics
5. New Instance Analysis (with normalization)
6. Temporal Patterns & Rolling Statistics
7. Spike Detection
8. ACF/PACF Analysis for Forecastability
9. Correlation Analysis

10. ML-Ready Feature Engineering
11. Autoscaler-Ready Summary

1.4 0. Setup & Imports

[ ]:
```
import polars as pl
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import acf, pacf
import warnings

warnings.filterwarnings('ignore')

# Set plotting style
plt.style.use('seaborn-v0_8-darkgrid')
sns.set_palette("husl")
plt.rcParams['figure.figsize'] = (14, 6)
plt.rcParams['font.size'] = 10

# Create output directory for summary tables
output_dir = Path("eda_summary")
output_dir.mkdir(exist_ok=True)

print(" Imports complete")
print(f" Output directory: {output_dir.absolute()}")
```

1.5 1. Data Loading & Schema Verification

[ ]:
```
# Load datasets using Polars
print(" Loading datasets...")

ml = pl.read_parquet("../data/processed/machine_level.parquet")
cl = pl.read_parquet("../data/processed/cluster_level.parquet")

print(f" Machine-level data: {len(ml):,} rows")
print(f" Cluster-level data: {len(cl):,} rows")
print("\n" + "="*80)
print("CLUSTER-LEVEL SCHEMA")
print("="*80)
print(cl.schema)
print("\n" + "="*80)
print("MACHINE-LEVEL SCHEMA")
print("="*80)
```

```
print(ml.schema)
```

[ ]:
```
# Display sample data
print("Cluster-Level Sample (first 10 rows):")
display(cl.head(10))

print("\nMachine-Level Sample (first 10 rows):")
display(ml.head(10))
```

1.6 2. Data Quality & Coverage Analysis

Critical: Assess data completeness, missing buckets, and machine coverage.

[ ]:
```
# Create bucket_index for temporal analysis
# bucket_s starts at 300, so bucket_index = (bucket_s / 300) - 1
cl = cl.with_columns([
    ((pl.col('bucket_s') / 300) - 1).cast(pl.Int64).alias('bucket_index')
])

ml = ml.with_columns([
    ((pl.col('bucket_s') / 300) - 1).cast(pl.Int64).alias('bucket_index')
])

print(" Created bucket_index for temporal analysis")
print(f"   Bucket index range: {cl['bucket_index'].min()} to␣
 ↪{cl['bucket_index'].max()}")
print(f"   Expected buckets: {cl['bucket_index'].max() - cl['bucket_index'].
 ↪min() + 1}")
print(f"   Actual buckets: {len(cl)}")
```

[ ]:
```
# Check for missing buckets
min_idx = cl['bucket_index'].min()
max_idx = cl['bucket_index'].max()
expected_buckets = set(range(min_idx, max_idx + 1))
actual_buckets = set(cl['bucket_index'].to_list())
missing_buckets = expected_buckets - actual_buckets

print("="*80)
print("DATA COVERAGE ANALYSIS")
print("="*80)
print(f"Expected bucket count: {len(expected_buckets):,}")
print(f"Actual bucket count:   {len(actual_buckets):,}")
print(f"Missing buckets:       {len(missing_buckets):,}")

if len(missing_buckets) > 0:
    print(f"\n   WARNING: {len(missing_buckets)} buckets are missing!")
```

```
    print(f"   Missing bucket indices (first 20):␣
 ↪{sorted(list(missing_buckets))[:20]}")
else:
    print("\n  No missing buckets - complete temporal coverage")
```

[ ]:
```
# Machine coverage analysis
total_unique_machines = ml['machine_id'].n_unique()
machines_per_bucket = ml.group_by('bucket_index').agg([
    pl.col('machine_id').n_unique().alias('unique_machines'),
    pl.count().alias('records')
]).sort('bucket_index')

print("\n" + "="*80)
print("MACHINE COVERAGE ANALYSIS")
print("="*80)
print(f"Total unique machines across all buckets: {total_unique_machines:,}")
print(f"\nMachines per bucket statistics:")
print(machines_per_bucket['unique_machines'].describe())

# Check for buckets with very low machine counts
low_coverage = machines_per_bucket.filter(pl.col('unique_machines') < 100)
if len(low_coverage) > 0:
    print(f"\n   WARNING: {len(low_coverage)} buckets have < 100 machines␣
 ↪reporting")
    print("   This may indicate data quality issues.")
else:
    print("\n  All buckets have adequate machine coverage")
```

[ ]:
```
# Plot machine coverage over time
fig, axes = plt.subplots(2, 1, figsize=(16, 10))

# Machines reporting per bucket
cl_sorted = cl.sort('bucket_index')
axes[0].plot(cl_sorted['bucket_index'], cl_sorted['machines'], linewidth=1,␣
 ↪alpha=0.7, color='steelblue')
axes[0].set_xlabel('Bucket Index')
axes[0].set_ylabel('Machines Reporting Usage')
axes[0].set_title('Number of Machines Reporting Usage Over Time')
axes[0].grid(True, alpha=0.3)
axes[0].axhline(cl_sorted['machines'].mean(), color='red', linestyle='--',␣
 ↪linewidth=2,
                label=f'Mean: {cl_sorted["machines"].mean():.0f}')
axes[0].legend()

# Records per bucket (usage density)
axes[1].plot(machines_per_bucket['bucket_index'],␣
 ↪machines_per_bucket['records'],
```

```
            linewidth=1, alpha=0.7, color='darkorange')
axes[1].set_xlabel('Bucket Index')
axes[1].set_ylabel('Number of Usage Records')
axes[1].set_title('Usage Record Density Over Time')
axes[1].grid(True, alpha=0.3)
axes[1].axhline(machines_per_bucket['records'].mean(), color='red',␣
 ↪linestyle='--', linewidth=2,
               label=f'Mean: {machines_per_bucket["records"].mean():.0f}')
axes[1].legend()

plt.tight_layout()
plt.show()

print("\n Interpretation:")
print("   - 'machines' column represents machines reporting usage, NOT total␣
 ↪capacity")
print("   - Variations indicate dynamic cluster behavior")
print("   - Sparse coverage may indicate partial data or machine churn")
```

1.7 3. Temporal Coverage & Basic Statistics

Using bucket_index for all temporal analysis (NOT real timestamps)

[ ]:
```
# Temporal coverage
bucket_interval_s = 300  # 5 minutes
total_duration_s = (cl['bucket_index'].max() + 1) * bucket_interval_s
total_duration_hours = total_duration_s / 3600
total_duration_days = total_duration_hours / 24

print("="*80)
print("TEMPORAL COVERAGE (using bucket_index)")
print("="*80)
print(f"Bucket interval:     {bucket_interval_s} seconds (5 minutes)")
print(f"Total buckets:       {cl['bucket_index'].max() + 1:,}")
print(f"Total duration:      {total_duration_hours:.1f} hours␣
 ↪({total_duration_days:.1f} days)")
print(f"Bucket index range:  {cl['bucket_index'].min()} to {cl['bucket_index'].
 ↪max()}")

print("\n" + "="*80)
print("DEMAND STATISTICS")
print("="*80)
print(f"CPU Demand:    min={cl['cpu_demand'].min():.2f}, max={cl['cpu_demand'].
 ↪max():.2f}, "
      f"mean={cl['cpu_demand'].mean():.2f}, std={cl['cpu_demand'].std():.2f}")
print(f"Memory Demand: min={cl['mem_demand'].min():.2f}, max={cl['mem_demand'].
 ↪max():.2f}, "
```

```
      f"mean={cl['mem_demand'].mean():.2f}, std={cl['mem_demand'].std():.2f}")
print(f"\nMachine Count: min={cl['machines'].min():,}, max={cl['machines'].
 ↪max():,}, "
      f"mean={cl['machines'].mean():.0f}, std={cl['machines'].std():.0f}")
print(f"New Instances: total={cl['new_instances_cluster'].sum():,}, "
      f"mean_per_bucket={cl['new_instances_cluster'].mean():.2f}")
```

1.8 4. Demand Distribution & Statistics

[ ]:
```
# Convert to pandas for plotting
cl_pd = cl.sort('bucket_index').to_pandas()

fig, axes = plt.subplots(2, 2, figsize=(16, 10))

# CPU demand distribution
axes[0, 0].hist(cl_pd['cpu_demand'], bins=100, edgecolor='black', alpha=0.7,␣
 ↪color='steelblue')
axes[0, 0].axvline(cl_pd['cpu_demand'].mean(), color='red', linestyle='--',␣
 ↪linewidth=2,
                   label=f'Mean: {cl_pd["cpu_demand"].mean():.2f}')
axes[0, 0].axvline(cl_pd['cpu_demand'].median(), color='orange',␣
 ↪linestyle='--', linewidth=2,
                   label=f'Median: {cl_pd["cpu_demand"].median():.2f}')
axes[0, 0].set_xlabel('CPU Demand (cores)')
axes[0, 0].set_ylabel('Frequency')
axes[0, 0].set_title('CPU Demand Distribution')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

# Memory demand distribution
axes[0, 1].hist(cl_pd['mem_demand'], bins=100, edgecolor='black', alpha=0.7,␣
 ↪color='darkorange')
axes[0, 1].axvline(cl_pd['mem_demand'].mean(), color='red', linestyle='--',␣
 ↪linewidth=2,
                   label=f'Mean: {cl_pd["mem_demand"].mean():.2f}')
axes[0, 1].axvline(cl_pd['mem_demand'].median(), color='blue', linestyle='--',␣
 ↪linewidth=2,
                   label=f'Median: {cl_pd["mem_demand"].median():.2f}')
axes[0, 1].set_xlabel('Memory Demand (normalized units, likely GB)')
axes[0, 1].set_ylabel('Frequency')
axes[0, 1].set_title('Memory Demand Distribution')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# CPU demand over bucket_index
axes[1, 0].plot(cl_pd['bucket_index'], cl_pd['cpu_demand'], linewidth=1,␣
 ↪alpha=0.7, color='steelblue')
```

```
axes[1, 0].set_xlabel('Bucket Index')
axes[1, 0].set_ylabel('CPU Demand (cores)')
axes[1, 0].set_title('CPU Demand Over Time (bucket_index)')
axes[1, 0].grid(True, alpha=0.3)

# Memory demand over bucket_index
axes[1, 1].plot(cl_pd['bucket_index'], cl_pd['mem_demand'], linewidth=1,␣
 ↪alpha=0.7, color='darkorange')
axes[1, 1].set_xlabel('Bucket Index')
axes[1, 1].set_ylabel('Memory Demand (normalized units)')
axes[1, 1].set_title('Memory Demand Over Time (bucket_index)')
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

[ ]:
```
# Percentile analysis
print("="*80)
print("DEMAND PERCENTILES")
print("="*80)
percentiles = [50, 75, 90, 95, 99]
for p in percentiles:
    cpu_p = np.percentile(cl_pd['cpu_demand'], p)
    mem_p = np.percentile(cl_pd['mem_demand'], p)
    print(f"P{p:2d}: CPU={cpu_p:8.2f} cores, Memory={mem_p:8.2f} units")

# Burstiness analysis
cpu_cv = cl_pd['cpu_demand'].std() / cl_pd['cpu_demand'].mean()
mem_cv = cl_pd['mem_demand'].std() / cl_pd['mem_demand'].mean()

print("\n" + "="*80)
print("BURSTINESS ANALYSIS (Coefficient of Variation)")
print("="*80)
print(f"CPU CV:    {cpu_cv:.4f}")
print(f"Memory CV: {mem_cv:.4f}")
print("\nInterpretation:")
print("  CV < 0.5: Low variability (predictable)")
print("  CV 0.5-1.0: Moderate variability")
print("  CV > 1.0: High variability (bursty, challenging for autoscaling)")
```

1.9 5. New Instance Analysis (with Normalization)

Critical: `new_instances_cluster` includes tasks that may NOT appear in usage data. We must normalize this metric.

[ ]:

```
# Normalize new instances by demand
cl_pd['new_instances_normalized'] = cl_pd['new_instances_cluster'] /␣
 ↪(cl_pd['cpu_demand'] + 1e-6)

print("="*80)
print("NEW INSTANCE ANALYSIS")
print("="*80)
print(f"Total new instances:           {cl_pd['new_instances_cluster'].sum():
 ↪,}")
print(f"Mean new instances per bucket: {cl_pd['new_instances_cluster'].mean():.
 ↪2f}")
print(f"Max new instances per bucket:  {cl_pd['new_instances_cluster'].max():
 ↪,}")
print(f"\nNormalized ratio (instances/demand):")
print(f"  Mean:   {cl_pd['new_instances_normalized'].mean():.2f}")
print(f"  Median: {cl_pd['new_instances_normalized'].median():.2f}")
print(f"  Max:    {cl_pd['new_instances_normalized'].max():.2f}")

# Detect extreme divergence
high_ratio = cl_pd[cl_pd['new_instances_normalized'] > 10000]
print(f"\n  Buckets with extreme instance/demand ratio (>10000):␣
 ↪{len(high_ratio)}")
if len(high_ratio) > 0:
    print("  This indicates instance_events include many tasks not reflected␣
 ↪in usage data.")
    print("  These must be normalized before use in ML or autoscaling.")
```

[ ]:
```
# Plot new instances vs demand
fig, axes = plt.subplots(2, 2, figsize=(16, 10))

# New instances over time
axes[0, 0].plot(cl_pd['bucket_index'], cl_pd['new_instances_cluster'],
                linewidth=1, alpha=0.7, color='purple')
axes[0, 0].set_xlabel('Bucket Index')
axes[0, 0].set_ylabel('New Instances')
axes[0, 0].set_title('New Instance Arrivals Over Time')
axes[0, 0].grid(True, alpha=0.3)

# Normalized ratio over time
axes[0, 1].plot(cl_pd['bucket_index'], cl_pd['new_instances_normalized'],
                linewidth=1, alpha=0.7, color='purple')
axes[0, 1].set_xlabel('Bucket Index')
axes[0, 1].set_ylabel('Instances / CPU Demand')
axes[0, 1].set_title('Normalized Instance Arrival Rate')
axes[0, 1].grid(True, alpha=0.3)
axes[0, 1].set_ylim(0, np.percentile(cl_pd['new_instances_normalized'], 99))  #␣
 ↪Cap at P99 for visibility
```

```
# Scatter: new instances vs CPU demand
sample_size = min(5000, len(cl_pd))
cl_sample = cl_pd.sample(n=sample_size, random_state=42)
axes[1, 0].scatter(cl_sample['cpu_demand'], cl_sample['new_instances_cluster'],
                   alpha=0.3, s=10, color='purple')
axes[1, 0].set_xlabel('CPU Demand (cores)')
axes[1, 0].set_ylabel('New Instances')
axes[1, 0].set_title(f'New Instances vs CPU Demand (n={sample_size:,})')
axes[1, 0].grid(True, alpha=0.3)

# Distribution of normalized ratio
axes[1, 1].hist(cl_pd['new_instances_normalized'], bins=100, edgecolor='black',
                alpha=0.7, color='purple', range=(0, np.
 ↪percentile(cl_pd['new_instances_normalized'], 99)))
axes[1, 1].set_xlabel('Instances / CPU Demand')
axes[1, 1].set_ylabel('Frequency')
axes[1, 1].set_title('Distribution of Normalized Instance Arrival Rate (capped␣
 ↪at P99)')
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\n Interpretation:")
print("   - Instance events and usage data are NOT perfectly aligned")
print("   - Many instance arrivals do not contribute to measured usage")
print("   - Normalization is essential before using this feature in ML models")
```

1.10 6. Temporal Patterns & Rolling Statistics

Using bucket_index for all temporal analysis

[ ]:
```
# Create synthetic time cycles for seasonality detection
# 12 buckets per hour (5min intervals), 288 buckets per day
buckets_per_hour = 12
buckets_per_day = 288

cl_pd['synthetic_hour'] = cl_pd['bucket_index'] % buckets_per_hour
cl_pd['synthetic_day'] = cl_pd['bucket_index'] % buckets_per_day

# Create rolling statistics
windows = {
    '1h': 12,   # 12 * 5min = 1 hour
    '6h': 72,   # 72 * 5min = 6 hours
    '24h': 288  # 288 * 5min = 24 hours
}
```

```
for name, window in windows.items():
    cl_pd[f'cpu_rolling_mean_{name}'] = cl_pd['cpu_demand'].
 ↪rolling(window=window, center=False).mean()
    cl_pd[f'cpu_rolling_std_{name}'] = cl_pd['cpu_demand'].
 ↪rolling(window=window, center=False).std()
    cl_pd[f'mem_rolling_mean_{name}'] = cl_pd['mem_demand'].
 ↪rolling(window=window, center=False).mean()
    cl_pd[f'mem_rolling_std_{name}'] = cl_pd['mem_demand'].
 ↪rolling(window=window, center=False).std()

print("  Created rolling statistics for 1h, 6h, 24h windows")
print("  Created synthetic time cycles for seasonality detection")
```

[ ]:
```
# Plot rolling statistics
fig, axes = plt.subplots(2, 1, figsize=(16, 10))

# CPU demand with rolling means
axes[0].plot(cl_pd['bucket_index'], cl_pd['cpu_demand'], linewidth=0.5, alpha=0.
 ↪3,
             color='gray', label='Raw Demand')
axes[0].plot(cl_pd['bucket_index'], cl_pd['cpu_rolling_mean_1h'], linewidth=1.5,
             alpha=0.8, color='blue', label='1h Rolling Mean')
axes[0].plot(cl_pd['bucket_index'], cl_pd['cpu_rolling_mean_6h'], linewidth=1.5,
             alpha=0.8, color='green', label='6h Rolling Mean')
axes[0].plot(cl_pd['bucket_index'], cl_pd['cpu_rolling_mean_24h'], linewidth=1.
 ↪5,
             alpha=0.8, color='red', label='24h Rolling Mean')
axes[0].set_xlabel('Bucket Index')
axes[0].set_ylabel('CPU Demand (cores)')
axes[0].set_title('CPU Demand with Rolling Means')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Memory demand with rolling means
axes[1].plot(cl_pd['bucket_index'], cl_pd['mem_demand'], linewidth=0.5, alpha=0.
 ↪3,
             color='gray', label='Raw Demand')
axes[1].plot(cl_pd['bucket_index'], cl_pd['mem_rolling_mean_1h'], linewidth=1.5,
             alpha=0.8, color='blue', label='1h Rolling Mean')
axes[1].plot(cl_pd['bucket_index'], cl_pd['mem_rolling_mean_6h'], linewidth=1.5,
             alpha=0.8, color='green', label='6h Rolling Mean')
axes[1].plot(cl_pd['bucket_index'], cl_pd['mem_rolling_mean_24h'], linewidth=1.
 ↪5,
             alpha=0.8, color='red', label='24h Rolling Mean')
axes[1].set_xlabel('Bucket Index')
```

```
axes[1].set_ylabel('Memory Demand')
axes[1].set_title('Memory Demand with Rolling Means')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

1.11 7. Spike Detection

Identify demand spikes using statistical thresholds

[ ]:
```
# Detect spikes (demand > rolling_mean + 3 * rolling_std)
threshold_sigma = 3

cl_pd['cpu_spike'] = (
    cl_pd['cpu_demand'] >
    (cl_pd['cpu_rolling_mean_6h'] + threshold_sigma *␣
 ↪cl_pd['cpu_rolling_std_6h'])
)

cl_pd['mem_spike'] = (
    cl_pd['mem_demand'] >
    (cl_pd['mem_rolling_mean_6h'] + threshold_sigma *␣
 ↪cl_pd['mem_rolling_std_6h'])
)

cpu_spikes = cl_pd[cl_pd['cpu_spike'] == True]
mem_spikes = cl_pd[cl_pd['mem_spike'] == True]

print("="*80)
print(f"SPIKE DETECTION (threshold: {threshold_sigma} above 6h rolling mean)")
print("="*80)
print(f"CPU spikes detected:    {len(cpu_spikes):,} ({len(cpu_spikes)/
 ↪len(cl_pd)*100:.2f}% of buckets)")
print(f"Memory spikes detected: {len(mem_spikes):,} ({len(mem_spikes)/
 ↪len(cl_pd)*100:.2f}% of buckets)")

if len(cpu_spikes) > 0:
    print(f"\nCPU spike statistics:")
    print(f"  Max spike magnitude: {cpu_spikes['cpu_demand'].max():.2f} cores")
    print(f"  Mean spike magnitude: {cpu_spikes['cpu_demand'].mean():.2f}␣
 ↪cores")

if len(mem_spikes) > 0:
    print(f"\nMemory spike statistics:")
    print(f"  Max spike magnitude: {mem_spikes['mem_demand'].max():.2f} units")
```

```
    print(f"  Mean spike magnitude: {mem_spikes['mem_demand'].mean():.2f}␣
 ↪units")
```

[ ]:
```
# Plot spikes
fig, axes = plt.subplots(2, 1, figsize=(16, 10))

# CPU spikes
axes[0].plot(cl_pd['bucket_index'], cl_pd['cpu_demand'], linewidth=0.5, alpha=0.
 ↪5,
             color='gray', label='CPU Demand')
axes[0].plot(cl_pd['bucket_index'], cl_pd['cpu_rolling_mean_6h'], linewidth=1.5,
             alpha=0.8, color='blue', label='6h Rolling Mean')
axes[0].plot(cl_pd['bucket_index'],
             cl_pd['cpu_rolling_mean_6h'] + threshold_sigma *␣
 ↪cl_pd['cpu_rolling_std_6h'],
             linewidth=1.5, alpha=0.8, color='red', linestyle='--',␣
 ↪label=f'{threshold_sigma} Threshold')
axes[0].scatter(cpu_spikes['bucket_index'], cpu_spikes['cpu_demand'],
                color='red', s=20, alpha=0.8, label=f'Spikes␣
 ↪(n={len(cpu_spikes)})', zorder=5)
axes[0].set_xlabel('Bucket Index')
axes[0].set_ylabel('CPU Demand (cores)')
axes[0].set_title('CPU Demand Spike Detection')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Memory spikes
axes[1].plot(cl_pd['bucket_index'], cl_pd['mem_demand'], linewidth=0.5, alpha=0.
 ↪5,
             color='gray', label='Memory Demand')
axes[1].plot(cl_pd['bucket_index'], cl_pd['mem_rolling_mean_6h'], linewidth=1.5,
             alpha=0.8, color='blue', label='6h Rolling Mean')
axes[1].plot(cl_pd['bucket_index'],
             cl_pd['mem_rolling_mean_6h'] + threshold_sigma *␣
 ↪cl_pd['mem_rolling_std_6h'],
             linewidth=1.5, alpha=0.8, color='red', linestyle='--',␣
 ↪label=f'{threshold_sigma} Threshold')
axes[1].scatter(mem_spikes['bucket_index'], mem_spikes['mem_demand'],
                color='red', s=20, alpha=0.8, label=f'Spikes␣
 ↪(n={len(mem_spikes)})', zorder=5)
axes[1].set_xlabel('Bucket Index')
axes[1].set_ylabel('Memory Demand')
axes[1].set_title('Memory Demand Spike Detection')
axes[1].legend()
axes[1].grid(True, alpha=0.3)
```

```
plt.tight_layout()
plt.show()

print("\n Interpretation:")
print("   - Spikes represent sudden demand increases that challenge reactive␣
 ↪autoscaling")
print("   - Proactive/predictive autoscaling can mitigate spike impact")
print("   - RL agents must learn to anticipate and handle these events")
```

1.12 8. ACF/PACF Analysis for Forecastability

Assess temporal autocorrelation to determine predictability

[ ]:
```
# Compute ACF and PACF for CPU demand
max_lags = 288  # 24 hours worth of lags

cpu_acf = acf(cl_pd['cpu_demand'].dropna(), nlags=max_lags, fft=True)
cpu_pacf_vals = pacf(cl_pd['cpu_demand'].dropna(), nlags=max_lags)

mem_acf = acf(cl_pd['mem_demand'].dropna(), nlags=max_lags, fft=True)
mem_pacf_vals = pacf(cl_pd['mem_demand'].dropna(), nlags=max_lags)

print(" Computed ACF/PACF for CPU and Memory demand")
```

[ ]:
```
# Plot ACF and PACF
fig, axes = plt.subplots(2, 2, figsize=(16, 10))

# CPU ACF
axes[0, 0].stem(range(len(cpu_acf)), cpu_acf, linefmt='steelblue',␣
 ↪markerfmt='o', basefmt=' ')
axes[0, 0].axhline(y=0, color='black', linewidth=0.8)
axes[0, 0].axhline(y=1.96/np.sqrt(len(cl_pd)), color='red', linestyle='--',␣
 ↪linewidth=1)
axes[0, 0].axhline(y=-1.96/np.sqrt(len(cl_pd)), color='red', linestyle='--',␣
 ↪linewidth=1)
axes[0, 0].set_xlabel('Lag (buckets)')
axes[0, 0].set_ylabel('ACF')
axes[0, 0].set_title('CPU Demand - Autocorrelation Function (ACF)')
axes[0, 0].grid(True, alpha=0.3)
axes[0, 0].set_xlim(0, max_lags)

# CPU PACF
axes[0, 1].stem(range(len(cpu_pacf_vals)), cpu_pacf_vals, linefmt='steelblue',␣
 ↪markerfmt='o', basefmt=' ')
axes[0, 1].axhline(y=0, color='black', linewidth=0.8)
axes[0, 1].axhline(y=1.96/np.sqrt(len(cl_pd)), color='red', linestyle='--',␣
 ↪linewidth=1)
```

```
axes[0, 1].axhline(y=-1.96/np.sqrt(len(cl_pd)), color='red', linestyle='--',␣
 ↪linewidth=1)
axes[0, 1].set_xlabel('Lag (buckets)')
axes[0, 1].set_ylabel('PACF')
axes[0, 1].set_title('CPU Demand - Partial Autocorrelation Function (PACF)')
axes[0, 1].grid(True, alpha=0.3)
axes[0, 1].set_xlim(0, max_lags)

# Memory ACF
axes[1, 0].stem(range(len(mem_acf)), mem_acf, linefmt='darkorange',␣
 ↪markerfmt='o', basefmt=' ')
axes[1, 0].axhline(y=0, color='black', linewidth=0.8)
axes[1, 0].axhline(y=1.96/np.sqrt(len(cl_pd)), color='red', linestyle='--',␣
 ↪linewidth=1)
axes[1, 0].axhline(y=-1.96/np.sqrt(len(cl_pd)), color='red', linestyle='--',␣
 ↪linewidth=1)
axes[1, 0].set_xlabel('Lag (buckets)')
axes[1, 0].set_ylabel('ACF')
axes[1, 0].set_title('Memory Demand - Autocorrelation Function (ACF)')
axes[1, 0].grid(True, alpha=0.3)
axes[1, 0].set_xlim(0, max_lags)

# Memory PACF
axes[1, 1].stem(range(len(mem_pacf_vals)), mem_pacf_vals, linefmt='darkorange',␣
 ↪markerfmt='o', basefmt=' ')
axes[1, 1].axhline(y=0, color='black', linewidth=0.8)
axes[1, 1].axhline(y=1.96/np.sqrt(len(cl_pd)), color='red', linestyle='--',␣
 ↪linewidth=1)
axes[1, 1].axhline(y=-1.96/np.sqrt(len(cl_pd)), color='red', linestyle='--',␣
 ↪linewidth=1)
axes[1, 1].set_xlabel('Lag (buckets)')
axes[1, 1].set_ylabel('PACF')
axes[1, 1].set_title('Memory Demand - Partial Autocorrelation Function (PACF)')
axes[1, 1].grid(True, alpha=0.3)
axes[1, 1].set_xlim(0, max_lags)

plt.tight_layout()
plt.show()
```

[ ]:
```
# Analyze forecastability
# Count significant lags (outside confidence interval)
confidence_bound = 1.96 / np.sqrt(len(cl_pd))
significant_cpu_acf = np.sum(np.abs(cpu_acf[1:]) > confidence_bound)
significant_mem_acf = np.sum(np.abs(mem_acf[1:]) > confidence_bound)

print("="*80)
```

```
print("FORECASTABILITY ANALYSIS")
print("="*80)
print(f"Significant ACF lags (CPU):    {significant_cpu_acf}/{max_lags} "
      f"({significant_cpu_acf/max_lags*100:.1f}%)")
print(f"Significant ACF lags (Memory): {significant_mem_acf}/{max_lags} "
      f"({significant_mem_acf/max_lags*100:.1f}%)")

# Analyze decay rate
cpu_acf_decay = np.where(cpu_acf[1:] < confidence_bound)[0]
mem_acf_decay = np.where(mem_acf[1:] < confidence_bound)[0]

if len(cpu_acf_decay) > 0:
    cpu_decay_lag = cpu_acf_decay[0] + 1
    print(f"\nCPU ACF decays to insignificance at lag {cpu_decay_lag} "
          f"({cpu_decay_lag * 5} minutes)")
else:
    print(f"\nCPU ACF remains significant throughout {max_lags} lags (strong␣
 ↪persistence)")

if len(mem_acf_decay) > 0:
    mem_decay_lag = mem_acf_decay[0] + 1
    print(f"Memory ACF decays to insignificance at lag {mem_decay_lag} "
          f"({mem_decay_lag * 5} minutes)")
else:
    print(f"Memory ACF remains significant throughout {max_lags} lags (strong␣
 ↪persistence)")

print("\n Interpretation:")
print("   - Strong ACF at multiple lags → demand is predictable")
print("   - Slow ACF decay → long-term dependencies exist")
print("   - This supports ML-based forecasting and proactive autoscaling")
print("   - RL agents can exploit temporal patterns for better decisions")
```

1.13 9. Correlation Analysis

[ ]:
```
# Correlation heatmap
corr_features = [
    'cpu_demand', 'mem_demand', 'machines', 'new_instances_normalized',
    'cpu_rolling_mean_6h', 'cpu_rolling_std_6h',
    'mem_rolling_mean_6h', 'mem_rolling_std_6h'
]

corr_matrix = cl_pd[corr_features].corr()

fig, ax = plt.subplots(figsize=(12, 10))
sns.heatmap(corr_matrix, annot=True, fmt='.3f', cmap='coolwarm', center=0,
            square=True, ax=ax, cbar_kws={'label': 'Correlation'})
```

```
ax.set_title('Feature Correlation Heatmap')
plt.tight_layout()
plt.show()

print("="*80)
print("KEY CORRELATIONS")
print("="*80)
print(f"CPU-Memory Correlation:          {cl_pd['cpu_demand'].
 ↪corr(cl_pd['mem_demand']):.4f}")
print(f"CPU-Machines Correlation:        {cl_pd['cpu_demand'].
 ↪corr(cl_pd['machines']):.4f}")
print(f"Memory-Machines Correlation:     {cl_pd['mem_demand'].
 ↪corr(cl_pd['machines']):.4f}")
print(f"CPU-NewInstances Correlation:    {cl_pd['cpu_demand'].
 ↪corr(cl_pd['new_instances_normalized']):.4f}")

print("\n Interpretation:")
print("   - High CPU-Memory correlation → resources scale together")
print("   - CPU-Machines correlation shows how well cluster tracked demand")
print("   - Low NewInstances correlation confirms events/usage mismatch")
```

1.14 10. ML-Ready Feature Engineering

Create a final dataset with engineered features for ML modeling

[ ]:
```
# Create lag features
lag_steps = [1, 5, 10, 20, 50, 100]

for lag in lag_steps:
    cl_pd[f'cpu_demand_lag{lag}'] = cl_pd['cpu_demand'].shift(lag)
    cl_pd[f'mem_demand_lag{lag}'] = cl_pd['mem_demand'].shift(lag)

print(f" Created lag features: {lag_steps}")
```

[ ]:
```
# Create cyclical features for synthetic time patterns
# These are SYNTHETIC and used ONLY for pattern detection
buckets_per_hour = 12
buckets_per_day = 288

cl_pd['sin_hour'] = np.sin(2 * np.pi * cl_pd['bucket_index'] / buckets_per_hour)
cl_pd['cos_hour'] = np.cos(2 * np.pi * cl_pd['bucket_index'] / buckets_per_hour)
cl_pd['sin_day'] = np.sin(2 * np.pi * cl_pd['bucket_index'] / buckets_per_day)
cl_pd['cos_day'] = np.cos(2 * np.pi * cl_pd['bucket_index'] / buckets_per_day)

print(" Created cyclical features (synthetic time patterns)")
print("   NOTE: These are NOT based on real timestamps, use for seasonality␣
 ↪detection only")
```

[ ]:
```
# Create normalized features
cl_pd['machines_normalized'] = (cl_pd['machines'] - cl_pd['machines'].mean()) /␣
 ↪cl_pd['machines'].std()
cl_pd['cpu_demand_normalized'] = (cl_pd['cpu_demand'] - cl_pd['cpu_demand'].
 ↪mean()) / cl_pd['cpu_demand'].std()
cl_pd['mem_demand_normalized'] = (cl_pd['mem_demand'] - cl_pd['mem_demand'].
 ↪mean()) / cl_pd['mem_demand'].std()

print("  Created normalized features (z-score)")
```

[ ]:
```
# Compile ML-ready dataset
ml_features = [
    'bucket_index',
    'cpu_demand', 'mem_demand',
    'machines', 'new_instances_normalized',
    'cpu_rolling_mean_1h', 'cpu_rolling_std_1h',
    'cpu_rolling_mean_6h', 'cpu_rolling_std_6h',
    'cpu_rolling_mean_24h', 'cpu_rolling_std_24h',
    'mem_rolling_mean_1h', 'mem_rolling_std_1h',
    'mem_rolling_mean_6h', 'mem_rolling_std_6h',
    'mem_rolling_mean_24h', 'mem_rolling_std_24h',
] + [f'cpu_demand_lag{lag}' for lag in lag_steps] + \
  [f'mem_demand_lag{lag}' for lag in lag_steps] + \
  ['sin_hour', 'cos_hour', 'sin_day', 'cos_day',
   'machines_normalized', 'cpu_demand_normalized', 'mem_demand_normalized']

ml_ready = cl_pd[ml_features].copy()

# Remove rows with NaN (due to rolling windows and lags)
ml_ready_clean = ml_ready.dropna()

print("="*80)
print("ML-READY DATASET")
print("="*80)
print(f"Total features:     {len(ml_features)}")
print(f"Total rows:         {len(ml_ready):,}")
print(f"Rows after dropna:  {len(ml_ready_clean):,}")
print(f"Rows dropped:       {len(ml_ready) - len(ml_ready_clean):,}")

print("\nFeature list:")
for i, feat in enumerate(ml_features, 1):
    print(f"  {i:2d}. {feat}")
```

[ ]:
```
# Display sample of ML-ready data
print("\nML-Ready Dataset Sample:")
display(ml_ready_clean.head(10))
```

```
print("\nML-Ready Dataset Statistics:")
display(ml_ready_clean.describe())
```

[ ]:
```
# Plot distributions of key engineered features
fig, axes = plt.subplots(3, 3, figsize=(18, 14))
axes = axes.flatten()

plot_features = [
    'cpu_demand_lag1', 'cpu_demand_lag10', 'cpu_demand_lag50',
    'cpu_rolling_mean_6h', 'cpu_rolling_std_6h',
    'sin_hour', 'cos_hour', 'sin_day', 'cos_day'
]

for i, feat in enumerate(plot_features):
    axes[i].hist(ml_ready_clean[feat].dropna(), bins=50, edgecolor='black',␣
 ↪alpha=0.7)
    axes[i].set_xlabel(feat)
    axes[i].set_ylabel('Frequency')
    axes[i].set_title(f'Distribution: {feat}')
    axes[i].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

[ ]:
```
# Save ML-ready dataset
output_path = output_dir / "ml_ready_features.csv"
ml_ready_clean.to_csv(output_path, index=False)
print(f"  Saved ML-ready dataset to: {output_path}")
print(f"   Shape: {ml_ready_clean.shape}")
```

1.15 11. Autoscaler-Ready Summary

Final summary of demand characteristics and autoscaling implications

[ ]:
```
# Compile summary statistics
summary = {
    'Temporal Coverage': {
        'Total Buckets': len(cl),
        'Duration (hours)': total_duration_hours,
        'Duration (days)': total_duration_days,
        'Bucket Interval': '5 minutes',
        'Missing Buckets': len(missing_buckets)
    },
    'Demand Statistics': {
        'CPU Mean': f"{cl_pd['cpu_demand'].mean():.2f} cores",
        'CPU Std': f"{cl_pd['cpu_demand'].std():.2f} cores",
        'CPU Max': f"{cl_pd['cpu_demand'].max():.2f} cores",
```

```
        'CPU P99': f"{np.percentile(cl_pd['cpu_demand'], 99):.2f} cores",
        'Memory Mean': f"{cl_pd['mem_demand'].mean():.2f} units",
        'Memory Std': f"{cl_pd['mem_demand'].std():.2f} units",
        'Memory Max': f"{cl_pd['mem_demand'].max():.2f} units",
        'Memory P99': f"{np.percentile(cl_pd['mem_demand'], 99):.2f} units"
    },
    'Burstiness': {
        'CPU CV': f"{cpu_cv:.4f}",
        'Memory CV': f"{mem_cv:.4f}",
        'CPU Spikes': f"{len(cpu_spikes)} ({len(cpu_spikes)/len(cl_pd)*100:.
↪2f}%)",
        'Memory Spikes': f"{len(mem_spikes)} ({len(mem_spikes)/len(cl_pd)*100:.
↪2f}%)"
    },
    'Predictability': {
        'CPU Significant ACF Lags': f"{significant_cpu_acf}/{max_lags}",
        'Memory Significant ACF Lags': f"{significant_mem_acf}/{max_lags}",
        'Strong Temporal Patterns': 'Yes' if significant_cpu_acf > max_lags * 0.
↪5 else 'Moderate'
    },
    'Instance Events': {
        'Total New Instances': f"{cl_pd['new_instances_cluster'].sum():,}",
        'Mean per Bucket': f"{cl_pd['new_instances_cluster'].mean():.2f}",
        'Events/Usage Alignment': 'Poor - requires normalization'
    },
    'Machine Coverage': {
        'Total Unique Machines': f"{total_unique_machines:,}",
        'Mean Machines per Bucket': f"{cl_pd['machines'].mean():.0f}",
        'Machine Count Range': f"{cl_pd['machines'].min():,} -␣
↪{cl_pd['machines'].max():,}"
    }
}

print("="*80)
print("AUTOSCALER-READY SUMMARY")
print("="*80)
for category, metrics in summary.items():
    print(f"\n{category}:")
    for key, value in metrics.items():
        print(f"  {key:30s}: {value}")
```

1.16 Key Findings & Autoscaling Implications

1.16.1 1. True Demand Characteristics

- **CPU demand** shows moderate to high variability (CV indicates burstiness)
- **Memory demand** follows similar but not identical patterns
- **P99 demand** significantly exceeds mean, requiring headroom in capacity planning

1.16.2 2. Burst Patterns

- Demand spikes detected using statistical thresholds (3 above rolling mean)
- Spikes represent sudden increases that challenge reactive autoscaling
- Proactive/predictive autoscaling can mitigate spike impact

1.16.3 3. Degree of Predictability

- **Strong autocorrelation** at multiple lags indicates temporal dependencies
- Slow ACF decay suggests long-term patterns exist
- This supports ML-based forecasting and proactive autoscaling strategies

1.16.4 4. Misalignment Between Usage and Instance Events

- `new_instances_cluster` includes tasks that may NOT appear in usage data
- Instance arrivals far exceed demand in many buckets
- **Critical:** Instance events must be normalized before use in ML or autoscaling

1.16.5 5. Expected Autoscaler Challenges

- **Reactive policies** will struggle with sudden spikes
- **Over-provisioning** needed to handle P99 scenarios
- **Under-utilization** likely during low-demand periods
- **Startup delays** for new machines compound latency

1.16.6 6. Reinforcement Learning Motivations

- RL can learn to **anticipate** demand patterns from temporal features
- RL can **balance** multiple objectives (cost, SLA, utilization)
- RL can **adapt** to non-stationary demand over time
- RL can exploit **long-term dependencies** revealed by ACF analysis

1.16.7 7. Why Capacity Must Be Simulated

- `machines` column represents machines **reporting usage**, NOT total capacity
- No ground truth for actual cluster capacity exists in the trace
- Simulator must model capacity decisions and constraints
- This allows testing different autoscaling policies against the same demand

---

1.17 Next Steps

1. **Baseline Autoscaling Simulation**
   - Implement threshold-based, target-tracking, and predictive policies
   - Measure SLA violations, cost, and utilization
2. **ML-Based Demand Forecasting**
   - Train LSTM/Transformer models on engineered features
   - Evaluate forecast accuracy at different horizons
   - Use forecasts for proactive scaling
3. **RL Agent Training**

- Design state space (demand, capacity, rolling stats, lags)
- Design action space (scale up/down by N machines)
- Design reward function (cost penalty + SLA violation penalty + utilization bonus)
- Train PPO/SAC agents

4. **Comparative Evaluation**
   - Benchmark all approaches on held-out test period
   - Analyze trade-offs and failure modes
   - Identify scenarios where each approach excels

END OF EDA