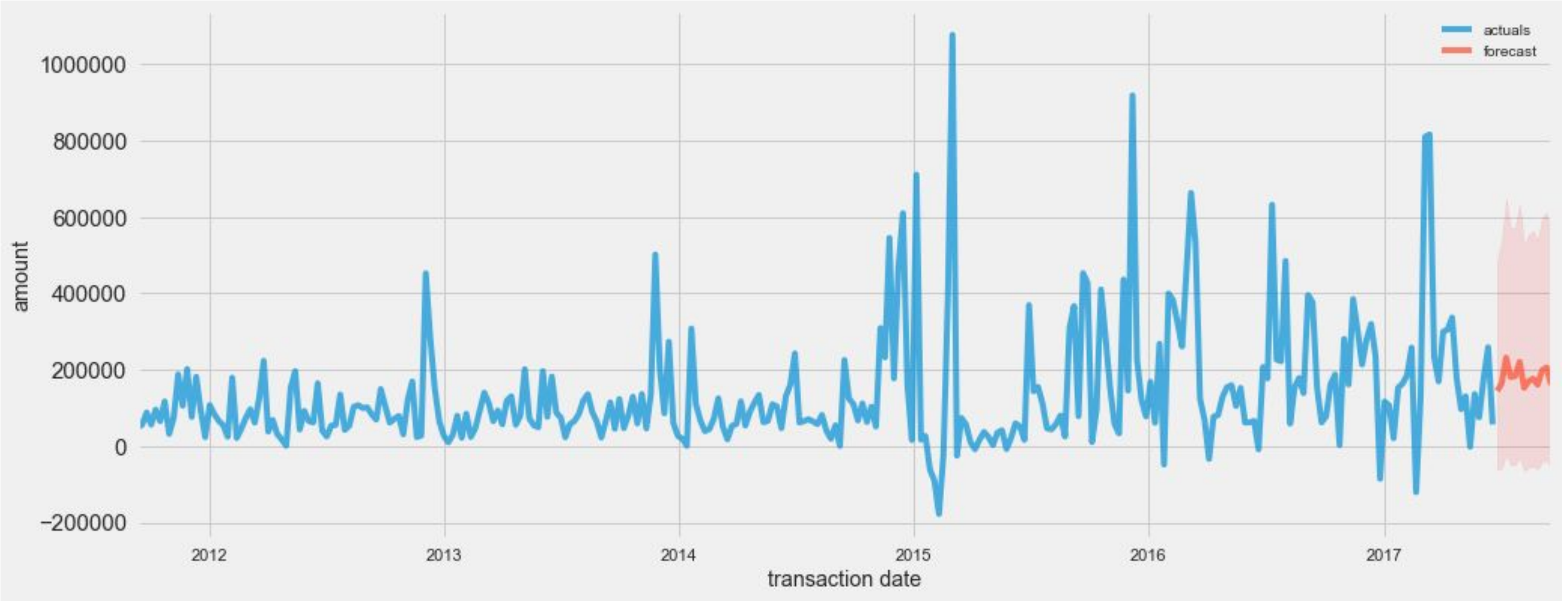
Robust Automated Forecasting In Python & R

Pranav Bahl, Data Scientist

Jonathan Stacks, DevOps Engineer

Time Series Forecasting

Time Series: A series of data points indexed in time order, spaced at equal time intervals. It consists of two variables, **time** and **values**.



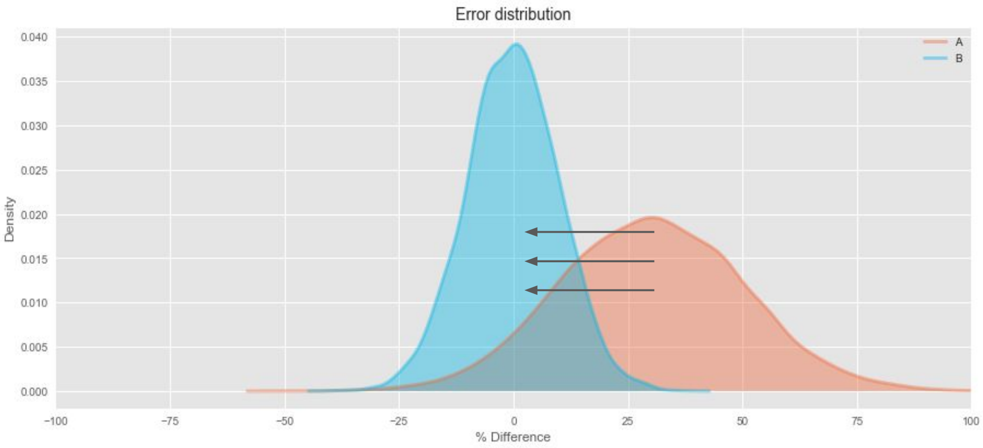
Overview

Goal: Demonstrate how to make millions of [robust] forecasts in an automated fashion

- Define the problem
- Retrieve and preprocess data
  - Profile
  - Transform
  - Detect outliers/anomalies
- Create forecast models employing multiple strategies and parameter combinations
  - Using Python
  - Using R
- Evaluate models with contextual evaluation metrics (and meta-metrics)
- Discuss how to choose the most appropriate hardware

Defining the problem

- Forecasts are used throughout our risk management system
  - Bias towards risk aversion
- > 250,000 unique time series
  - Growing at 2x each year
- Elastic compute capacity
- Runtime <= 5 hours
- Cost <= $200/day
- Reduce error by 10%
  - Based on cumulative forecast % error



Runtime and cost calculation

$$R = \left\lceil \frac{f_a * f_s * f_t}{c * s * \frac{60}{f_m}} \right\rceil$$

$$C = R * s * d$$

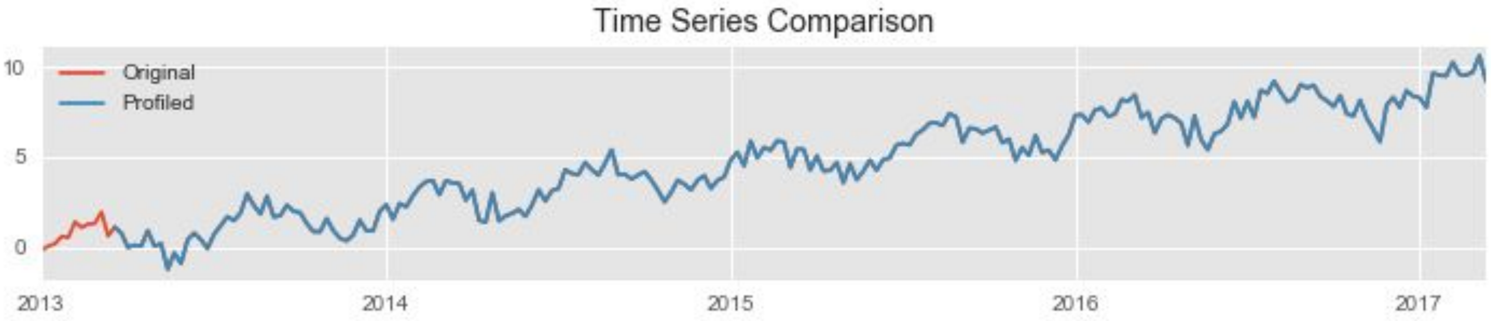
$R =$ Total Runtime of the Pipeline
$C =$ Cost to Run the Pipeline
$f_a =$ Number of Accounts to Process
$f_s =$ Number of Forecasting Strategies
$f_t =$ Number of Forecasting Transformations
$f_m =$ Mean Forecast Runtime
$s =$ Desired Number of Servers
$c =$ Cores per Server
$d =$ On Demand Cost per Hour

```
"""
ec2 cost estimation
"""

import math

# ec2 variables
s = 28.0        # desired server count
c = 35.0        # cores per server
d = 1.5         # on demand cost per hour

# forecasting variables
fa = 100000     # number of unique time series
fs = 4.0        # forecasting strategies
ft = 3.0        # forecasting transformations
fm = 0.33       # mean transformation/strategy runtime
fl = 5.00       # forecasting upper limit

# calculate runtime and cost
runtime = math.ceil((fa * fs * ft) / (c * s * (60.0 / fm)))
cost = runtime * s * d
```

Because the number of unique time series to be processed change over time so the idea is to adjust number of servers to get the job done in desired time.

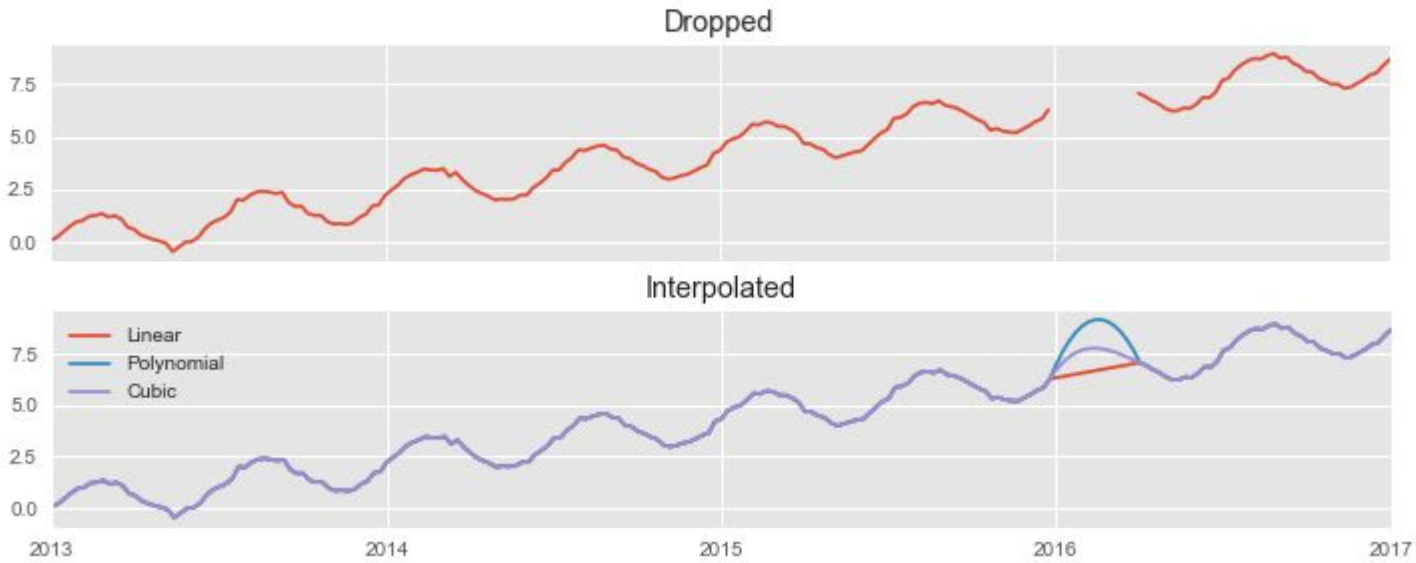
Time series profiling

- Ensure index is a timestamp
- Check for completeness of panel
- Remove leap days
- Make sure data has at least one complete season
- Truncate data to in favor of complete seasons



Handling missing values

- Impute using…
    - Descriptive statistics (e.g. mean or median)
    - Interpolation (e.g. linear or polynomial)
    - Extrapolation (e.g. training a linear model)



Outlier/anomaly detection

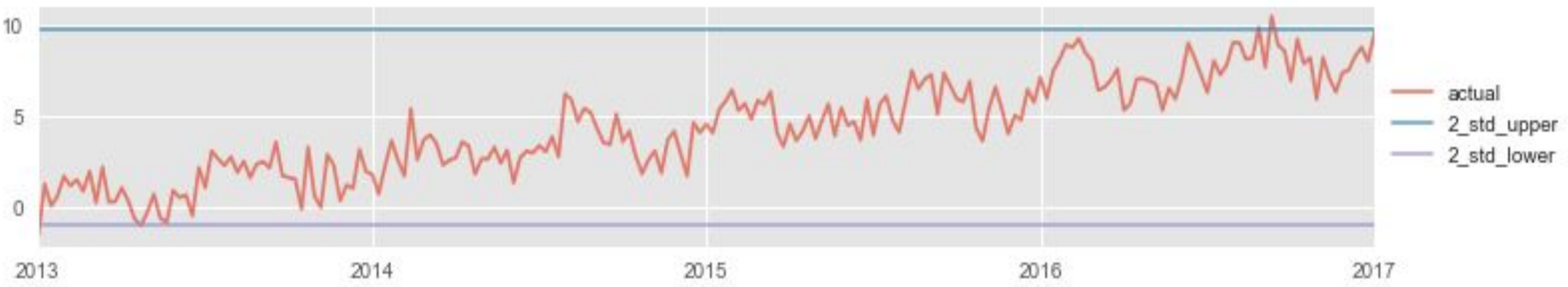
Outlier: An observation that lies an abnormal distance from other values in a random sample from a population.

- Boxplot
- Time-series decomposition routine

Anomaly: Illegitimate data point that’s generated by a different process than whatever generated the rest of the data.

- Forecasting
- Robust Principal Component Analysis (RPCA)

Outlier/anomaly detection



Forecasting library landscape

- Regression
  - [Python] StatsModels (OLS, polynomial)
  - [Python/R] XGBoost (gradient boosted regression trees)
- ARMA / ARIMA / SARMIA (Autoregressive Moving-Average)
  - [R] Forecast
  - [Python] StatsModels
  - [Python] Pyramid
- Exponential smoothing
  - [R] Forecast (Holt-Winters)
- Structural Models / State space models
  - [R] BSTS (Bayesian Structural Time Series)
  - [Python] Pyflux
  - [Python/R] Prophet

Evaluation Metrics

Evaluation metrics are used to measure the quality of forecast. It is also used to compare different strategies.There are two types of evaluation metrics:

Scale dependent: The metrics which requires all the compared time series be on the same scale.
Eg:

MAE(Mean Absolute Error)	$\mathbf{mean}(\lvert \mathbf{e_i} \rvert)$
MSE(Mean Squared Error)	$\mathbf{mean}(\mathbf{e_i^2})$
MFE(Mean Forecast Error)	$\mathbf{mean}(\mathbf{e_i})$
SSE(Sum of Squared Error)	$\sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{e_i^2}$
RMSE(Root Mean Squared Error)	$\sqrt{\mathbf{mean}(\mathbf{e_i^2})}$
SMSE(Signed Mean Squared Error)	$\frac{\mathbf{1}}{\mathbf{n}} \sum_{\mathbf{i=1}}^{\mathbf{n}} (\frac{\mathbf{e_i}}{\lvert \mathbf{e_i} \rvert}) \mathbf{e_i^2}$
NMSE(Normalised Mean Squared Error)	$\frac{\mathbf{MSE}}{\sigma^2}$

$$\mathbf{e_i} = \mathbf{y_i} - \hat{\mathbf{y}}_\mathbf{i}$$
$$\sigma = \frac{\mathbf{1}}{\mathbf{n-1}} \sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{e_i^2}$$

Scale independent: The metrics used to compare forecast performance amongst different data sets
Eg:

MPE(Mean Percentage Error)	$\mathbf{mean}(\frac{\mathbf{e_i}}{\hat{\mathbf{y}}}) * 100$
MAPE(Mean Absolute Percentage Error)	$\mathbf{mean}(\left\lvert \frac{\mathbf{e_i}}{\hat{\mathbf{y}}} \right\rvert) * 100$
MASE(Mean Absolute Scaled Error)	$\mathbf{mean}(\frac{\mathbf{e_i}}{\frac{\mathbf{1}}{\mathbf{I-1}} \sum_{\mathbf{i=2}}^{\mathbf{I}}} \lvert \mathbf{y_i} - \mathbf{y_{i-1}} \rvert)$

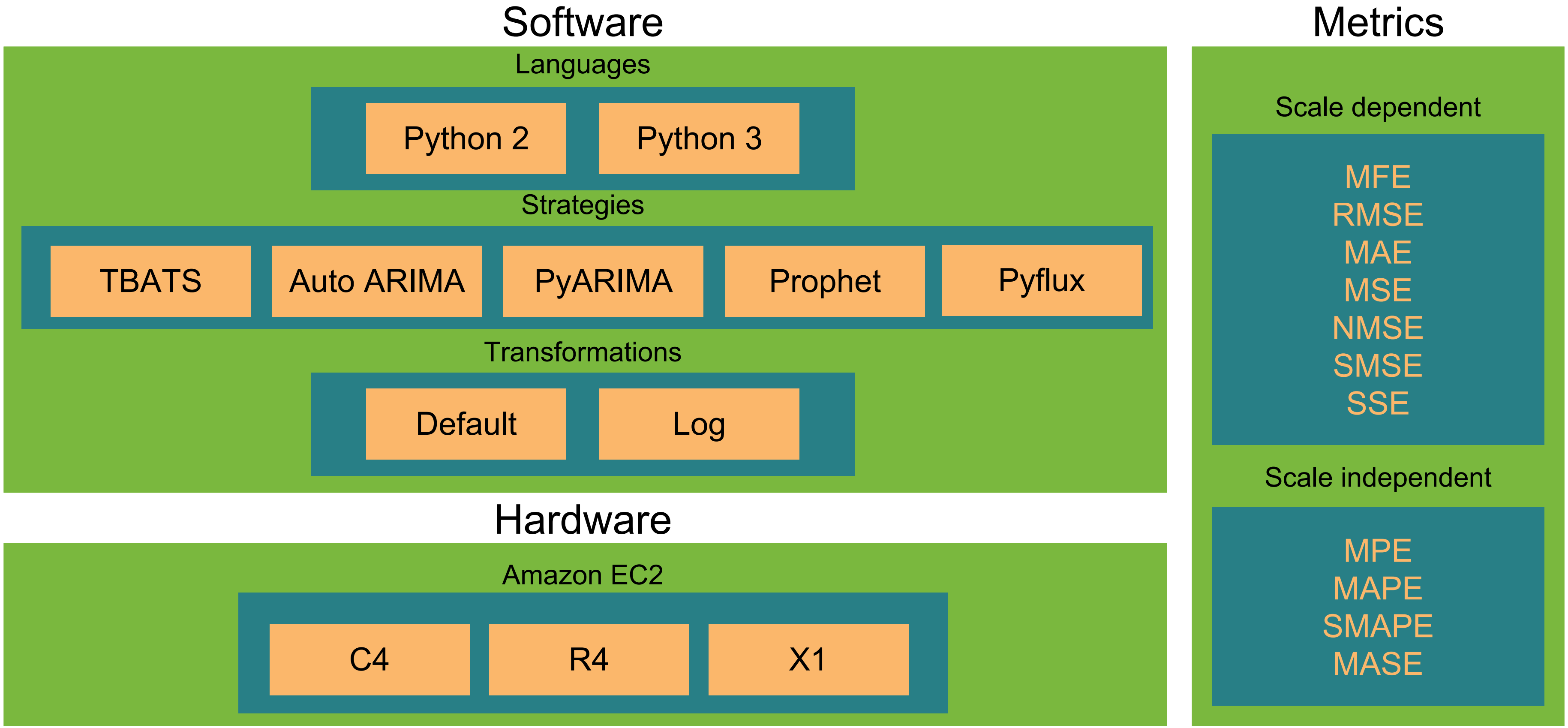
Bringing Down Runtime, Scale and Cost

Runtime optimization: Run different experiments to narrow down poor performing strategies or transformations

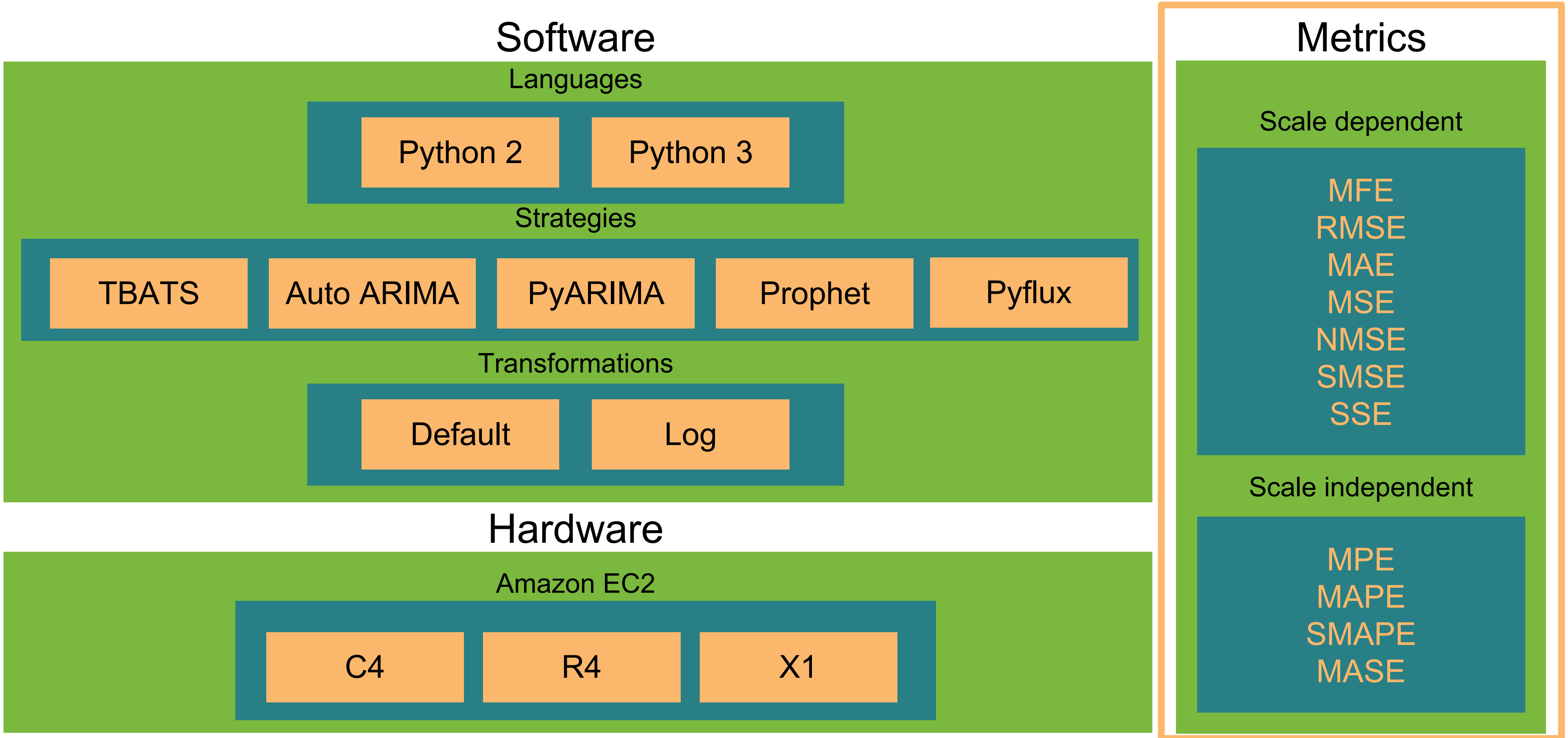
Cost optimization: Optimize the cost function by experimenting different server options

Runtime optimization

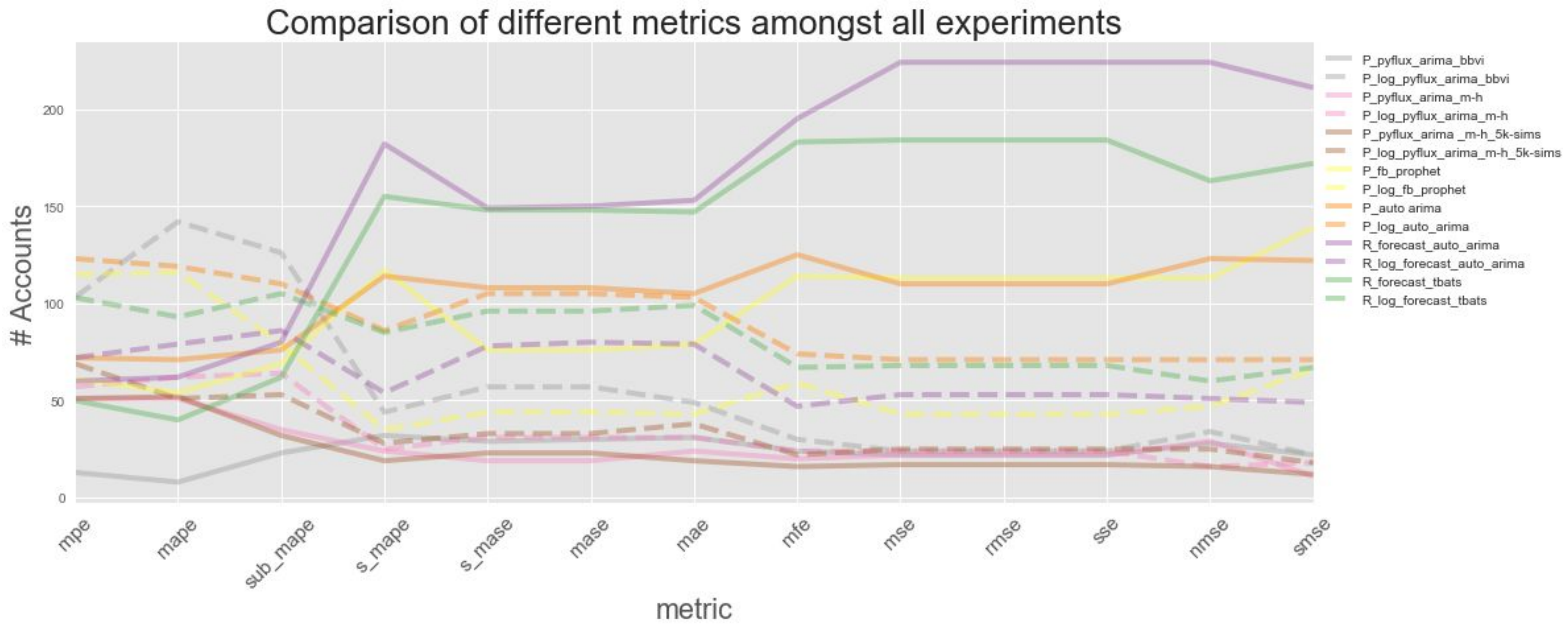
Initial Approach



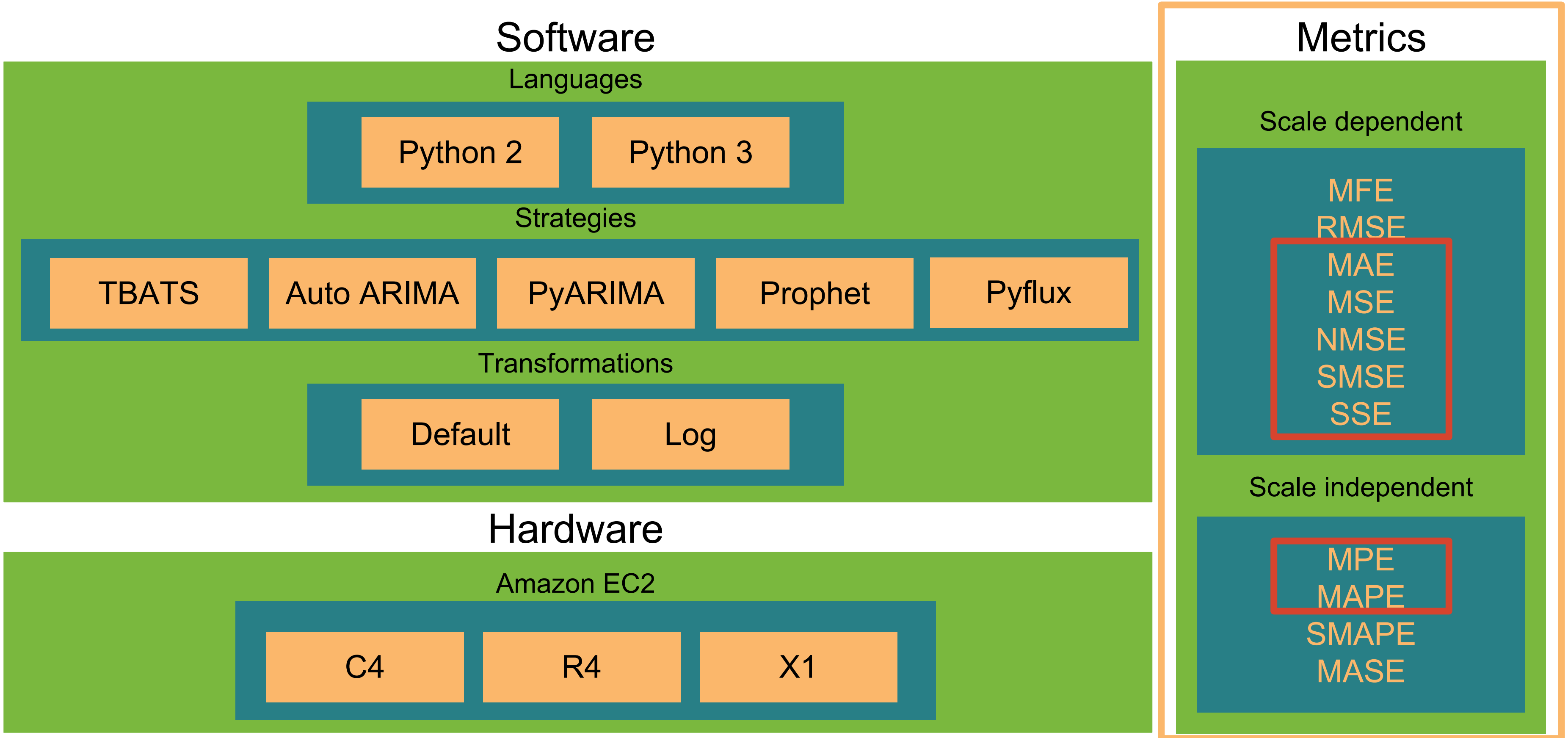
First experiment: Remove unnecessary metrics



First experiment: Remove unnecessary metrics



First experiment: Remove unnecessary metrics



First experiment: Result

Software

Languages

- Python 2
- Python 3

Strategies

- TBATS
- Auto ARIMA
- PyARIMA
- Prophet
- Pyflux

Transformations

- Default
- Log

Hardware

Amazon EC2

- C4
- R4
- X1

Metrics

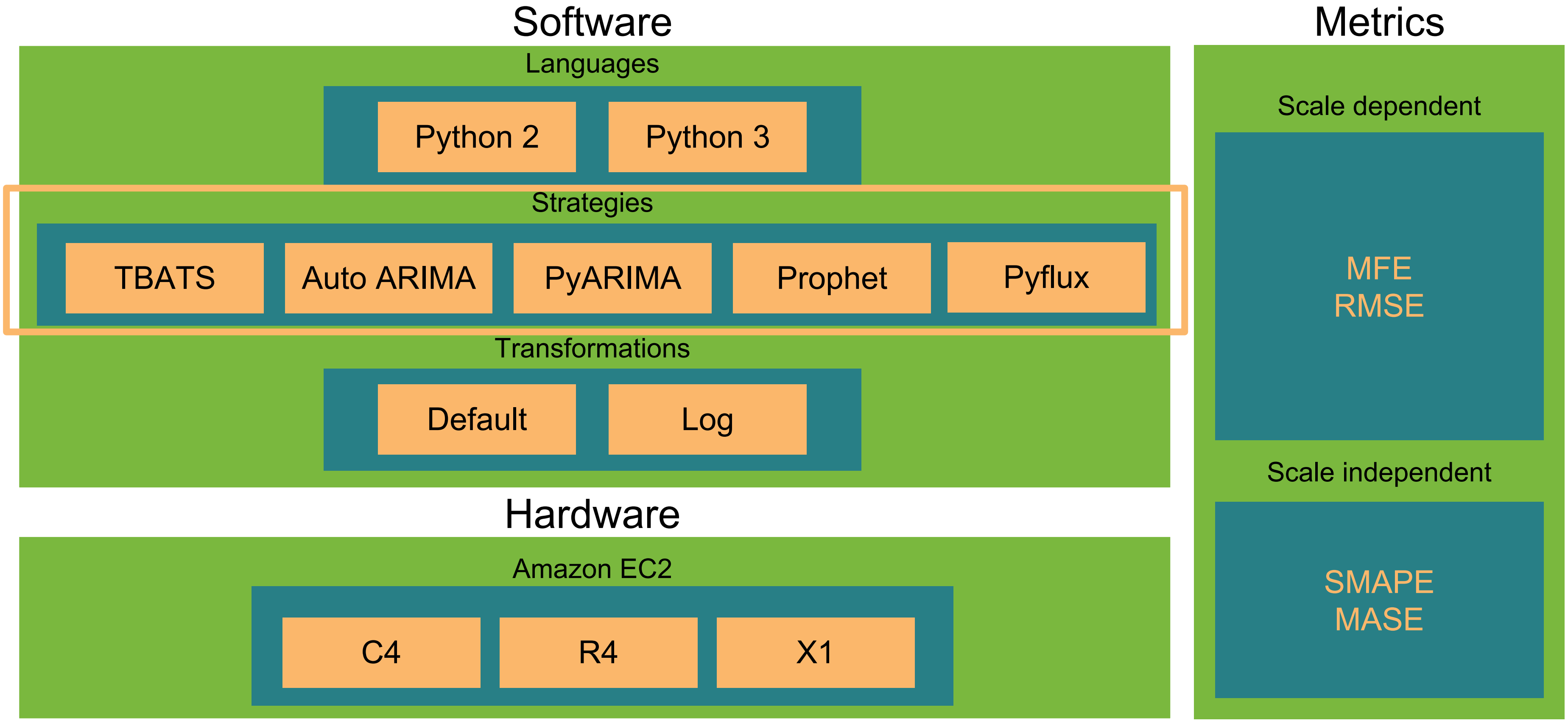
Scale dependent

- MFE
- RMSE

Scale independent

- SMAPE
- MASE

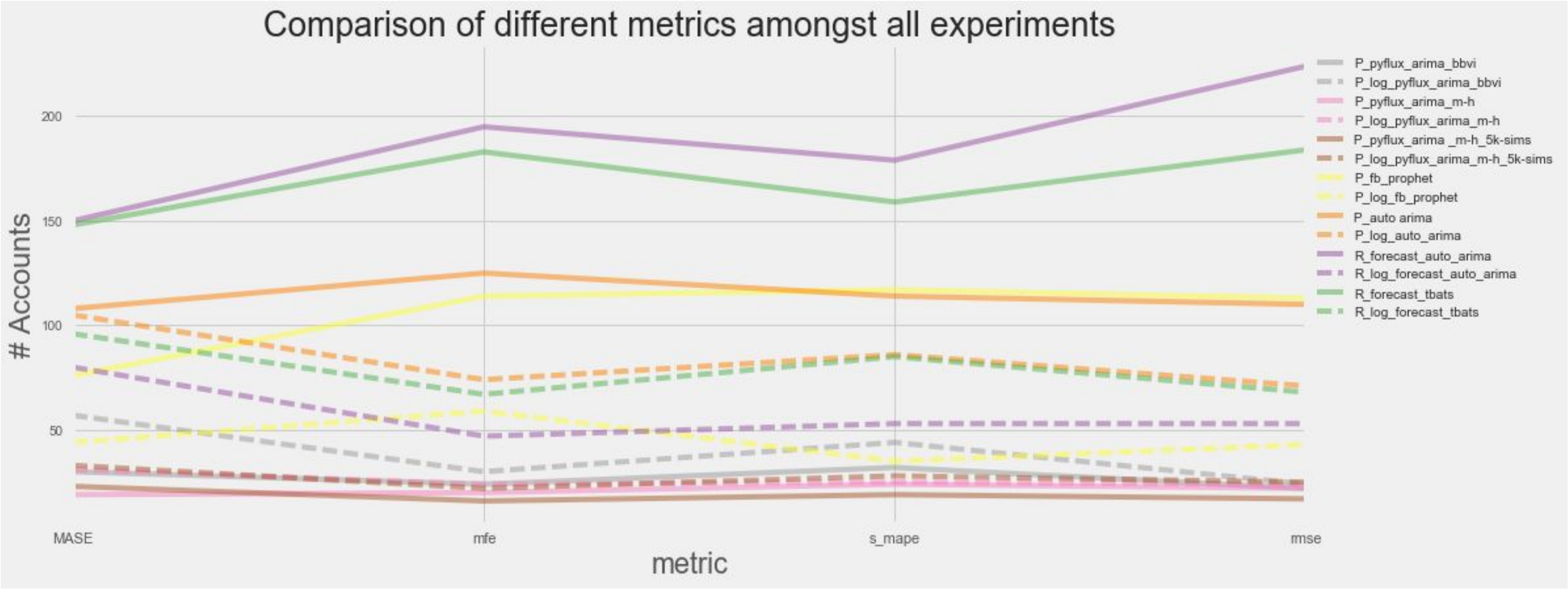
Second experiment: Forecast model selection



Second experiment: Forecast model selection

- Choose distinct but meaningful parameters for all strategies
- Initial experiment with default settings
  - Exception: reduced number of simulations for faster runtime
- Compare strategies across filtered evaluation metrics

Second experiment: Forecast model selection



Second experiment: Forecast model selection

Software

Languages

- Python 2
- Python 3

Strategies

- TBATS
- Auto ARIMA
- PyARIMA
- Prophet
- Pyflux

Transformations

- Default
- Log

Hardware

Amazon EC2

- C4
- R4
- X1

Metrics

Scale dependent

- MFE
- RMSE

Scale independent

- SMAPE
- MASE

Second experiment: Result

Software

Languages
- Python 2
- Python 3

Strategies
- TBATS
- Auto ARIMA
- PyARIMA
- Prophet

Transformations
- Default
- Log

Hardware

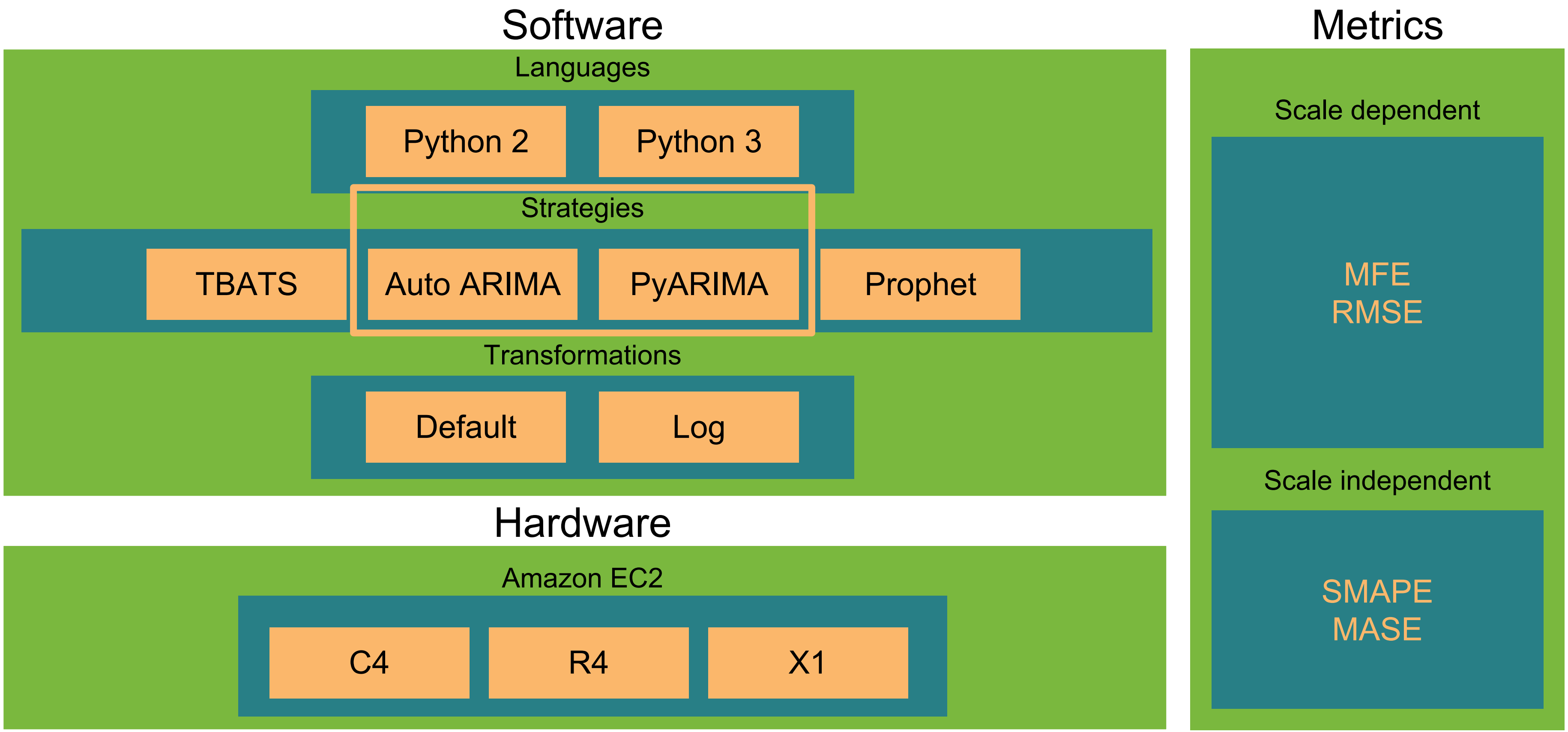
Amazon EC2
- C4
- R4
- X1

Metrics

Scale dependent
- MFE
- RMSE

Scale independent
- SMAPE
- MASE

Third experiment: Auto ARIMA vs PyARIMA



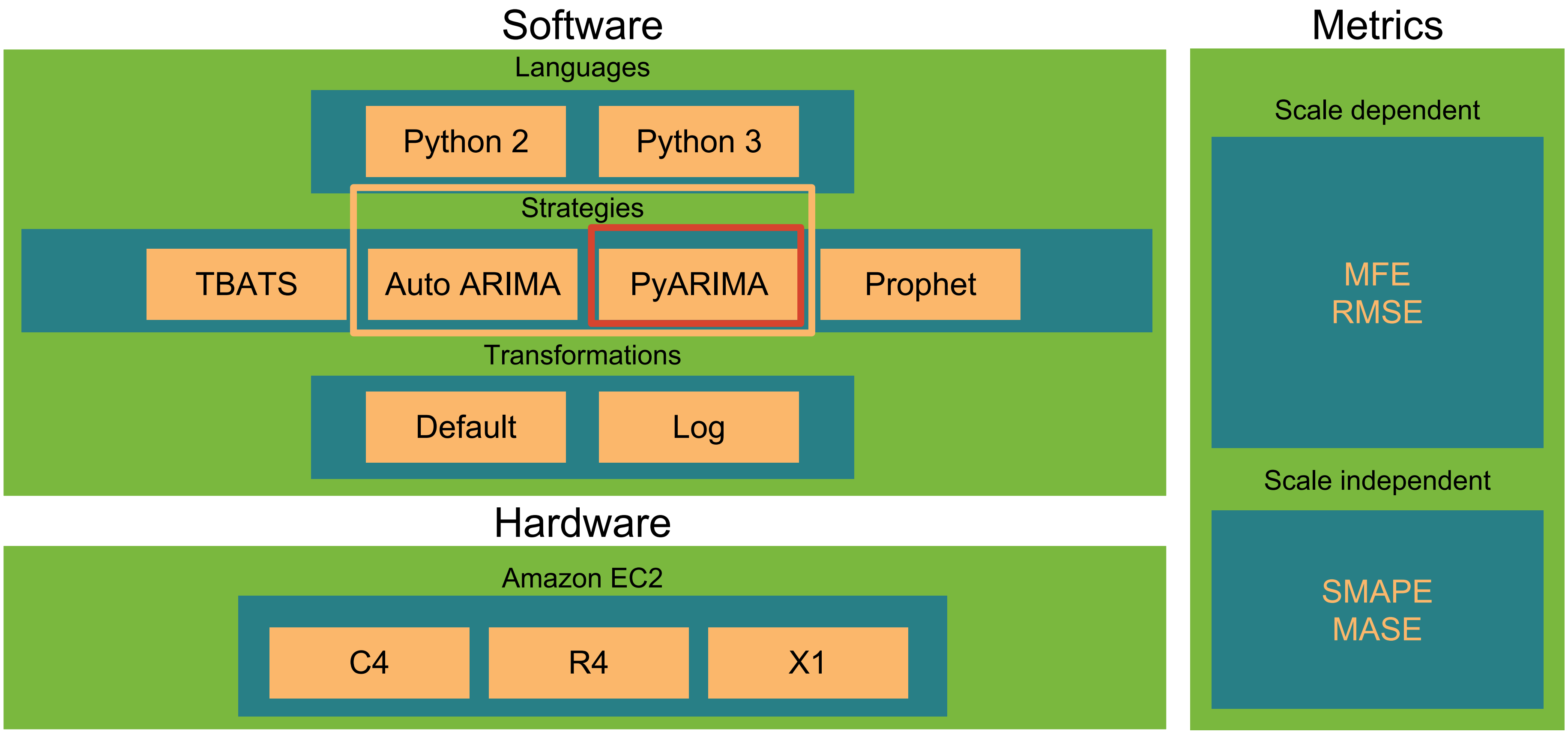
Third experiment: Auto ARIMA vs PyARIMA

- Choose best Arima Implementation
  - Since Python's Auto Arima reflect R's implementation
- Compare error differences
  - Hold implementation with better performance

Error differences, where Python Auto ARIMA wins

50% py_auto_arima	**mfe**	1367.300
	rmse	519.010
50% py_auto_arima_log	**mfe**	3731.060
	rmse	131.640

Third experiment: Auto ARIMA vs PyARIMA



Third experiment: Result

Software

Languages

- Python 2
- Python 3

Strategies

- TBATS
- Auto ARIMA
- Prophet

Transformations

- Default
- Log

Hardware

Amazon EC2

- C4
- R4
- X1

Metrics

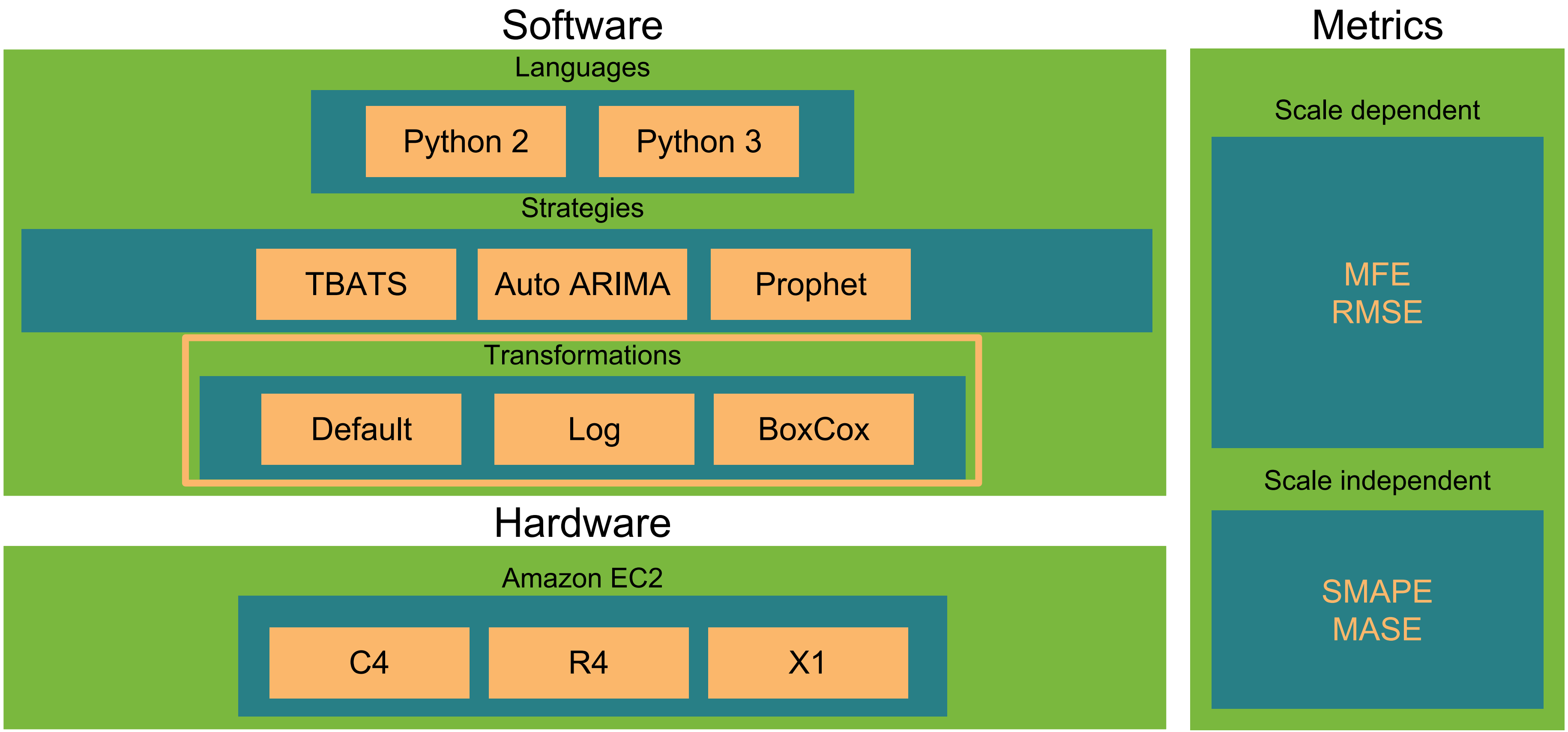
Scale dependent

- MFE
- RMSE

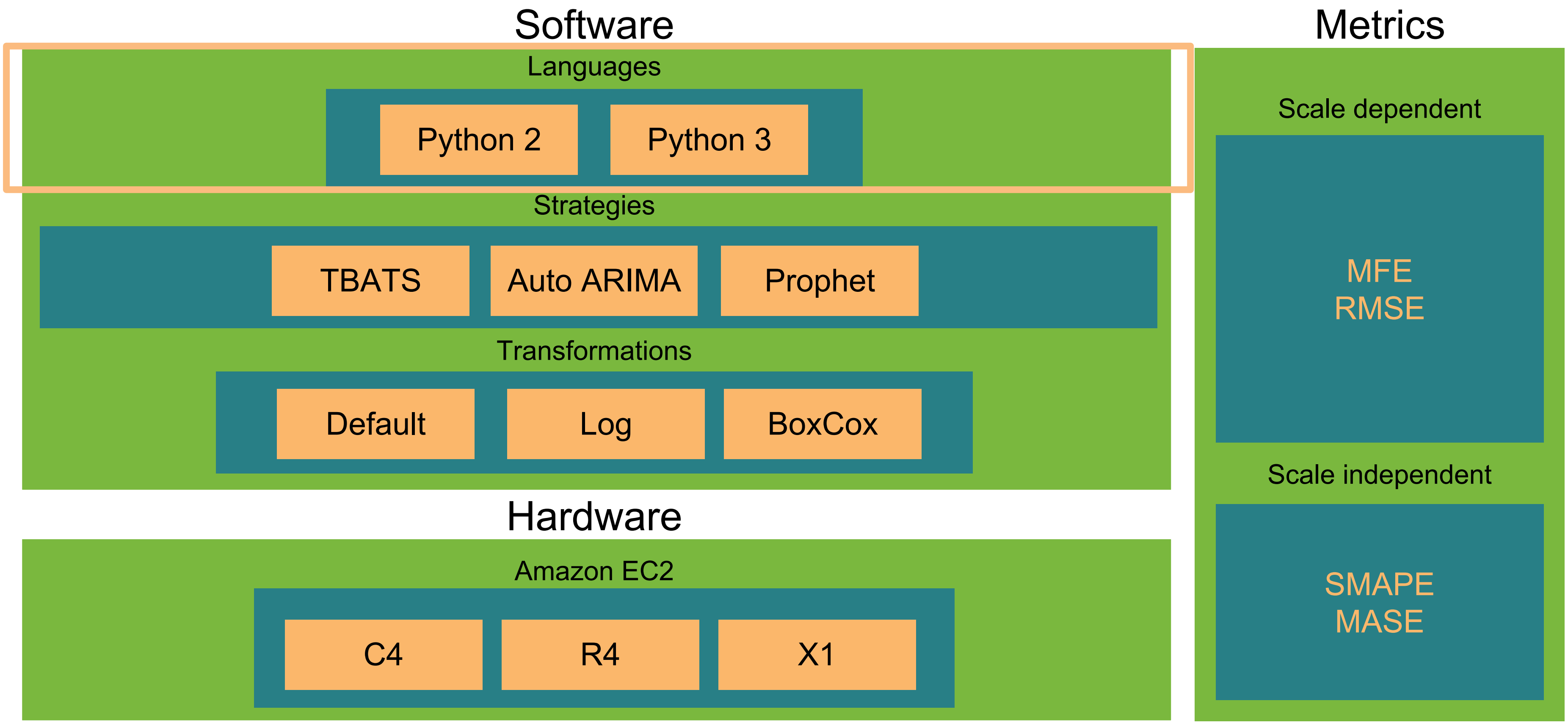
Scale independent

- SMAPE
- MASE

Added another transformation

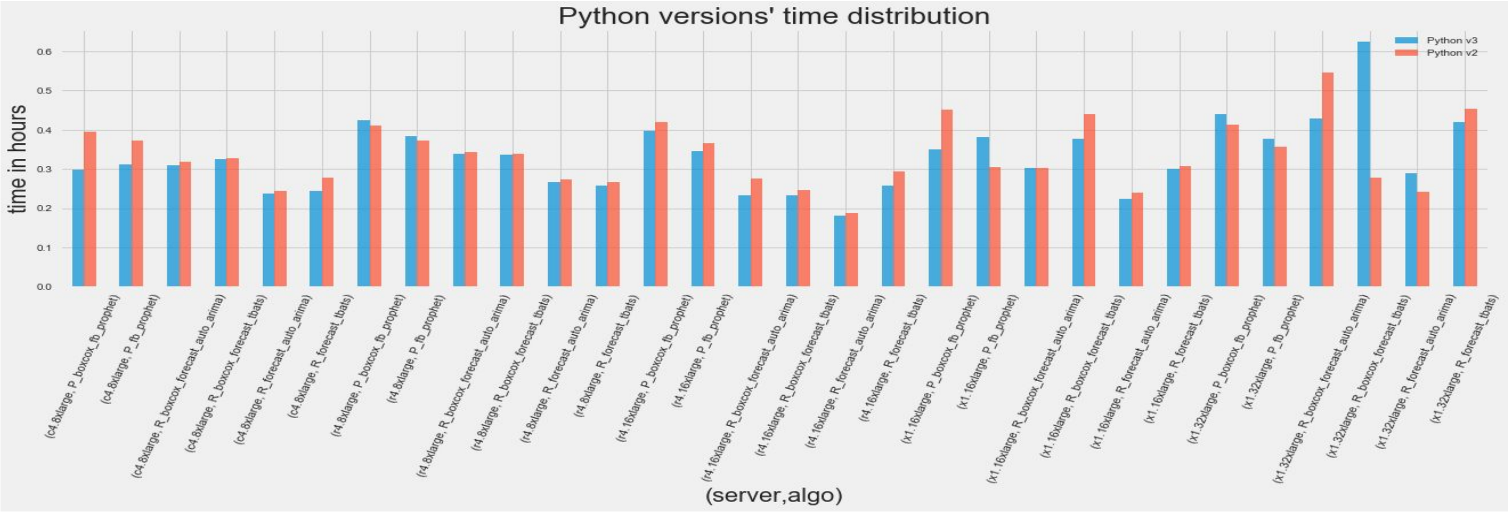


Fourth Experiment: Python Version Testing

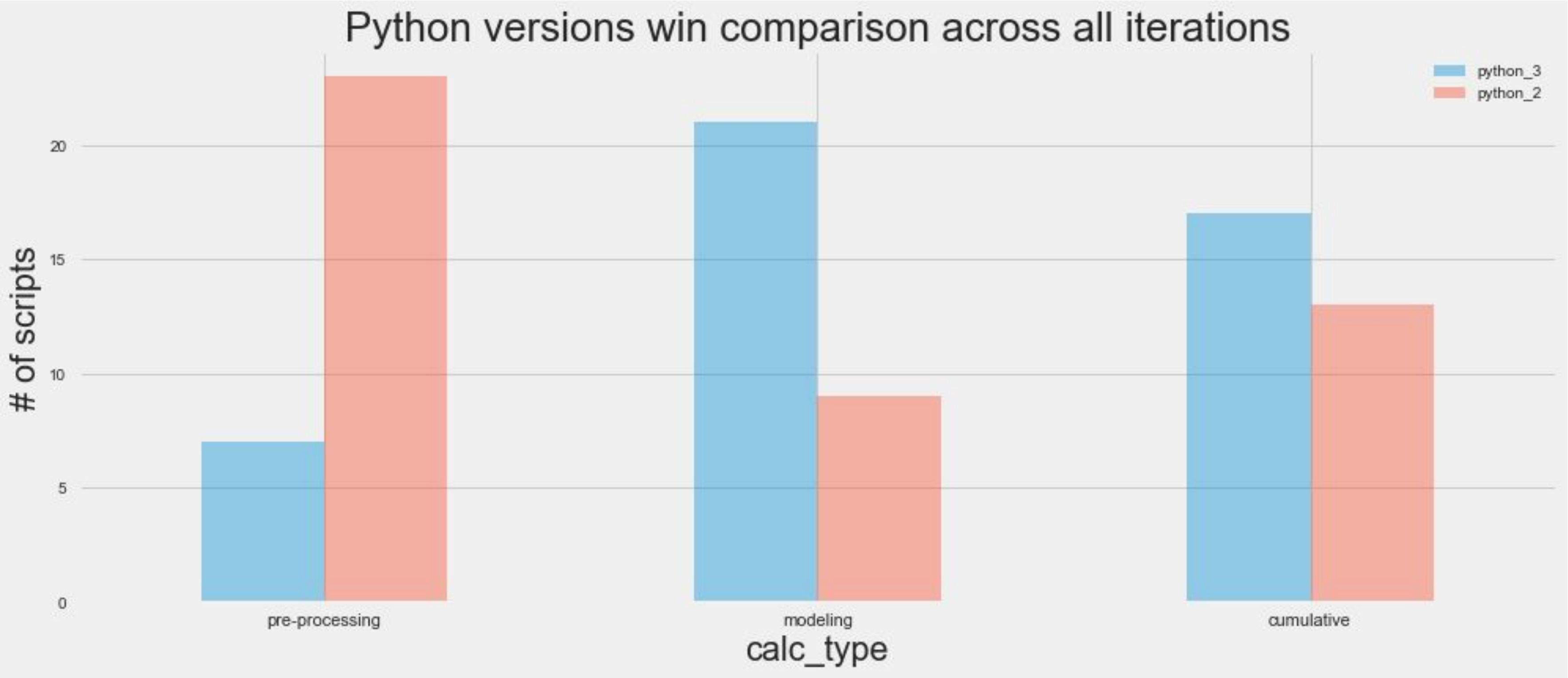


Fourth Experiment: Python Version Testing

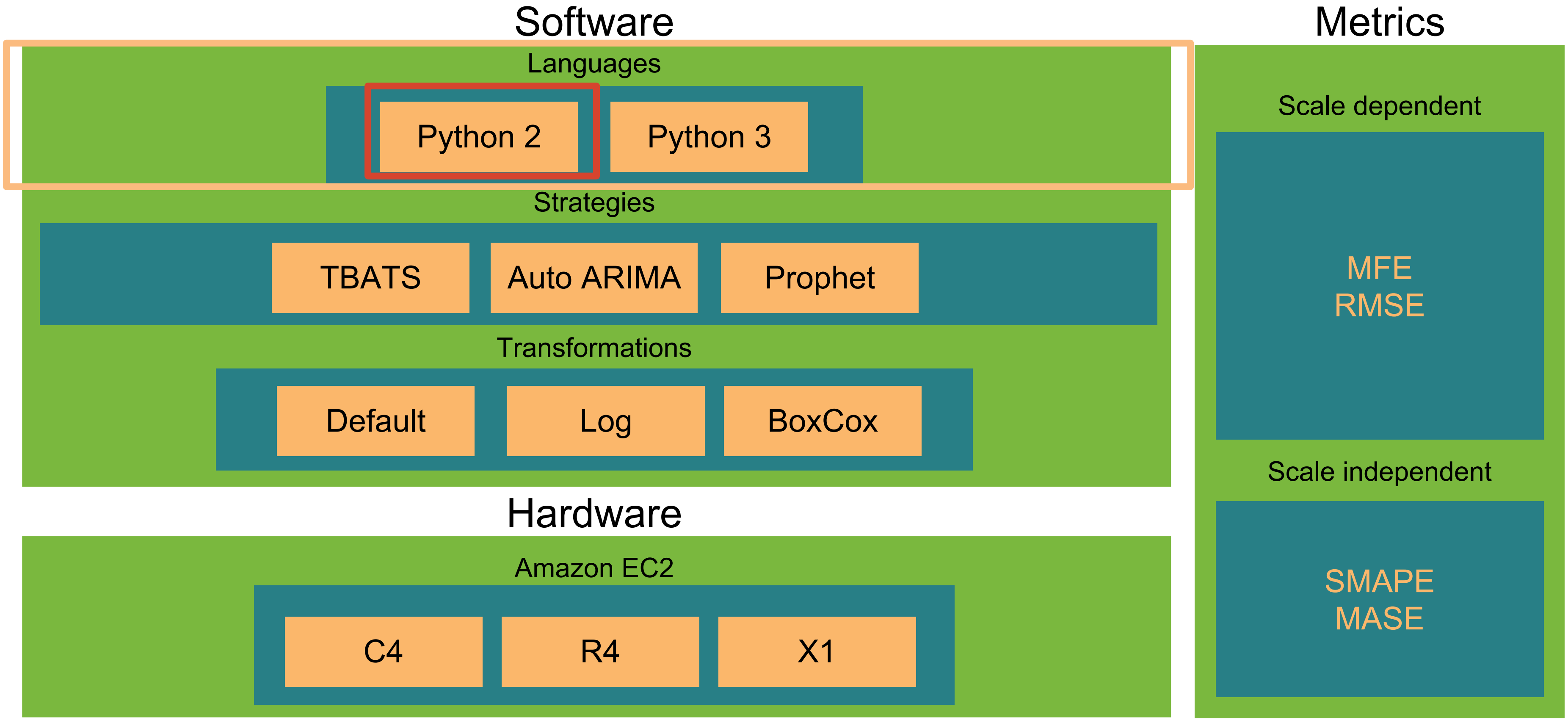
- Execute all strategies on both Python versions
  - Run over different server types
- Compare different processing times



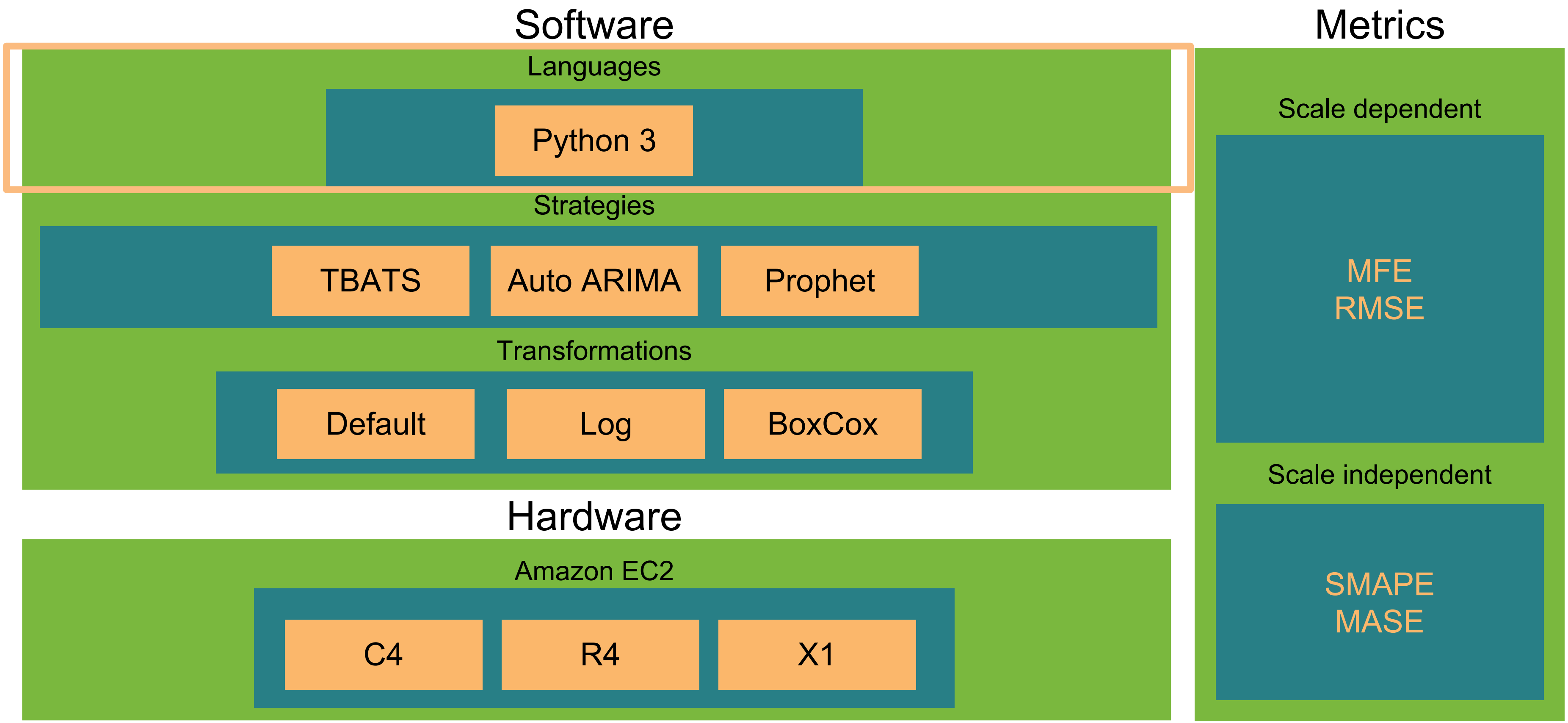
Fourth Experiment: Python Version Testing



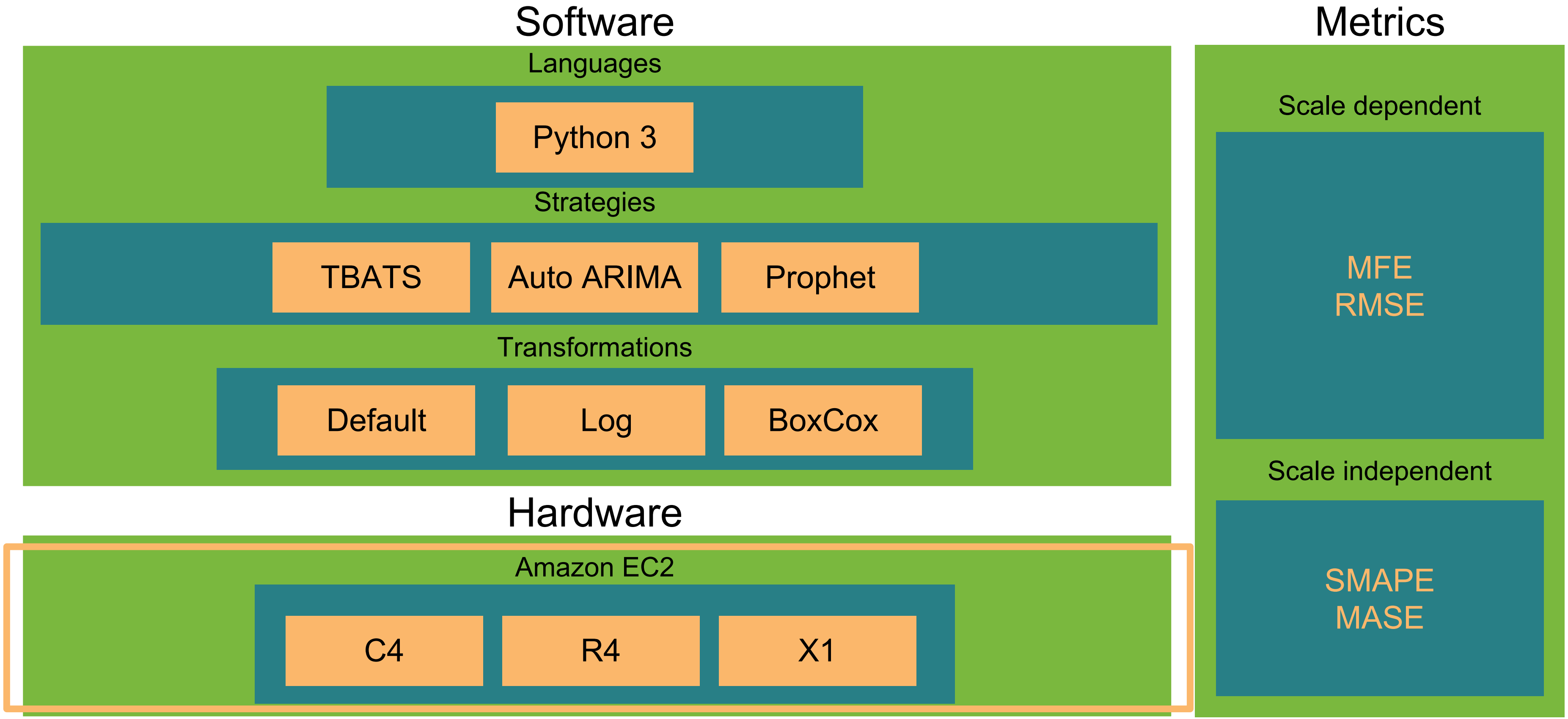
Fourth Experiment: Python Version Testing



Fourth Experiment: Result

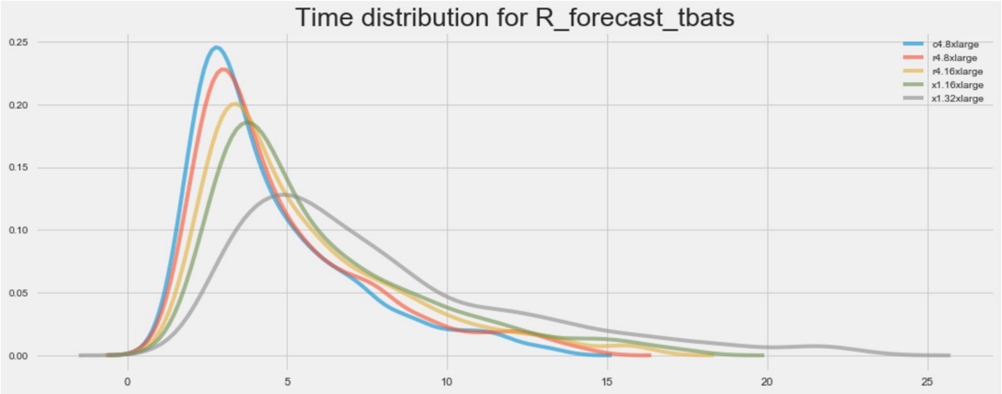


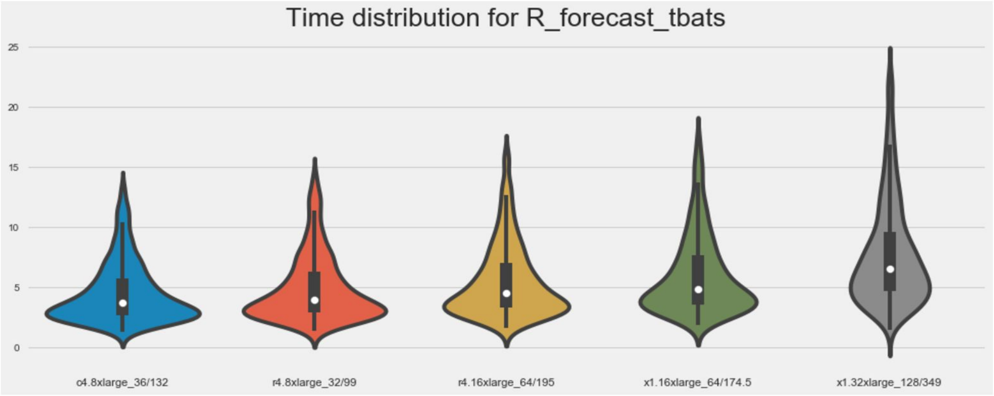
Fifth Experiment: Hardware Optimization



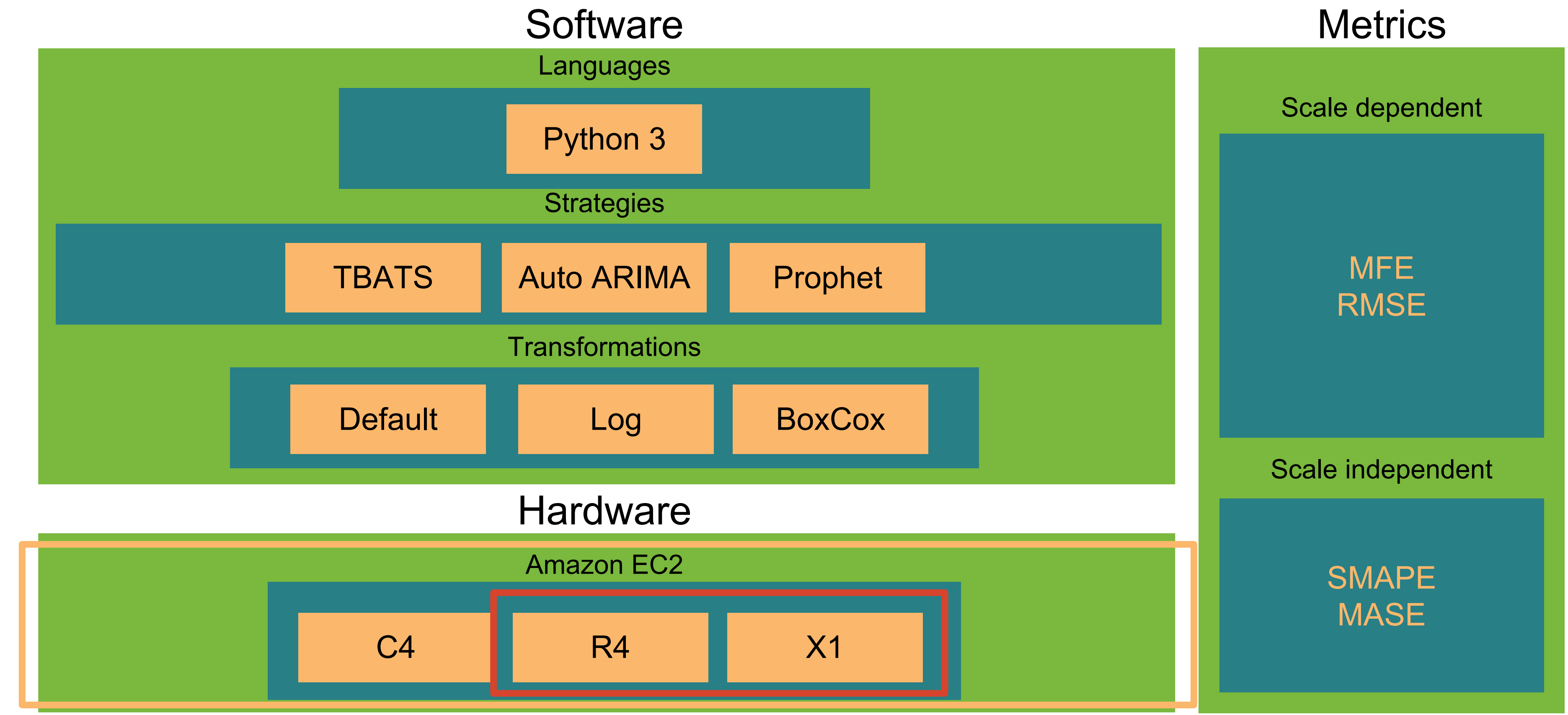
Fifth Experiment: Hardware Optimization

Goal: Run all strategies with different transformation across different servers. Pick best server based on time(cumulative) consumption.





Fifth Experiment: Hardware Optimization



Fifth Experiment: Result

Software

Languages
- Python 3

Strategies
- TBATS
- Auto ARIMA
- Prophet

Transformations
- Default
- Log
- BoxCox

Hardware

Amazon EC2
- C4

Metrics

Scale dependent
- MFE
- RMSE

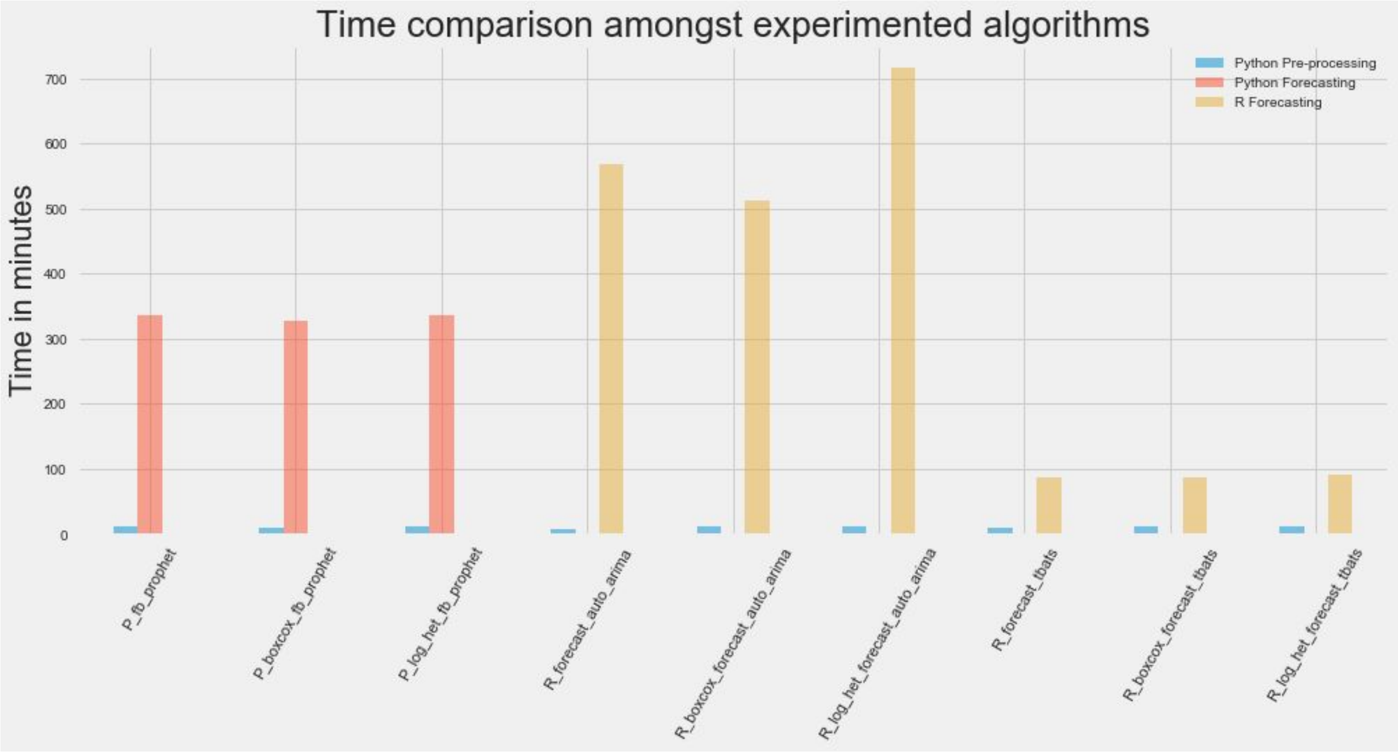
Scale independent
- SMAPE
- MASE

Sixth Experiment: How to improve processing time?

Determine bottlenecks:

- Determine if any strategy is consuming considerably more time compared to others
- Try optimizing overhead created by rpy2 while spinning up R kernel

Sixth Experiment: How to improve processing time?



Sixth Experiment: How to improve processing time?

Let’s look at the time distribution of different strategies

Across all samples

	Auto_ARIMA	TBATS	Prophet
	c4.8xlarge	c4.8xlarge	c4.8xlarge
count	1000.000	1000.000	1000.000
mean	33.891	5.189	3.684
std	99.507	4.068	0.422
min	0.070	1.280	1.890
25%	0.140	2.700	3.430
50%	0.955	3.770	3.670
75%	24.605	6.190	3.950
max	1601.560	31.400	5.480

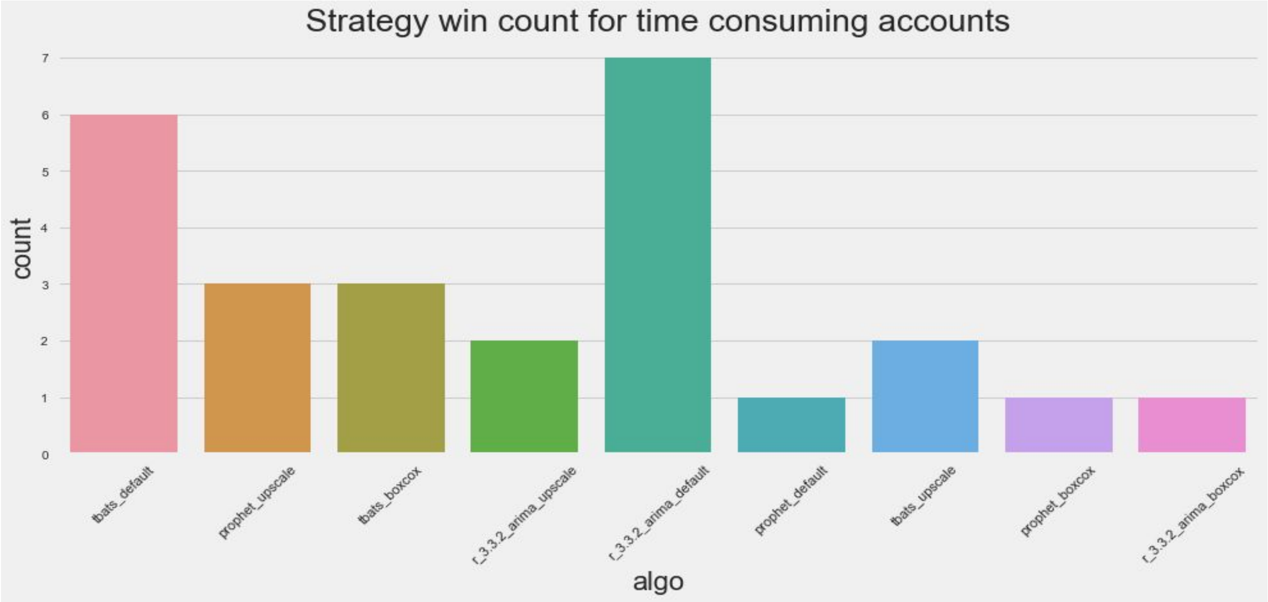
Under 2nd standard deviation

	Auto_ARIMA	TBATS	Prophet
	c4.8xlarge	c4.8xlarge	c4.8xlarge
count	963.000	955.000	958.000
mean	19.188	4.511	3.684
std	40.244	2.510	0.347
min	0.070	1.280	2.850
25%	0.130	2.655	3.440
50%	0.790	3.660	3.670
75%	22.845	5.750	3.928
max	232.570	13.290	4.520

Result: Apply timeout duration of 300 secs and analyse the loss of forecasting strength

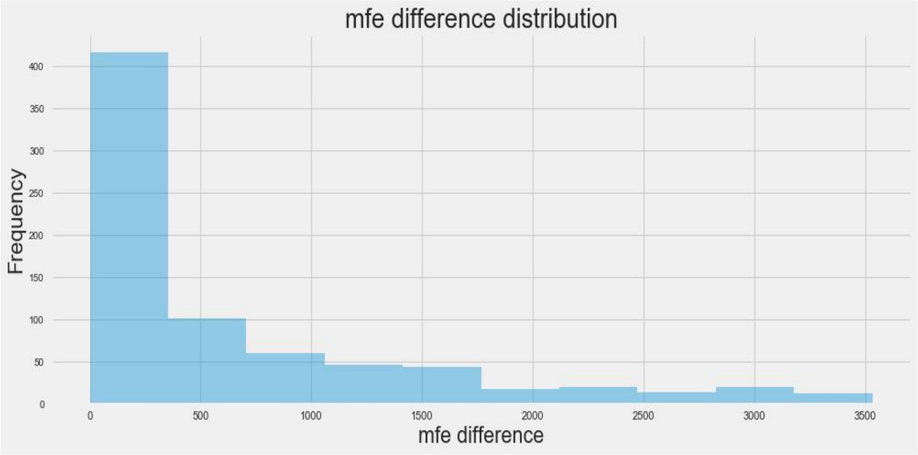
Sixth Experiment: How to improve processing time?

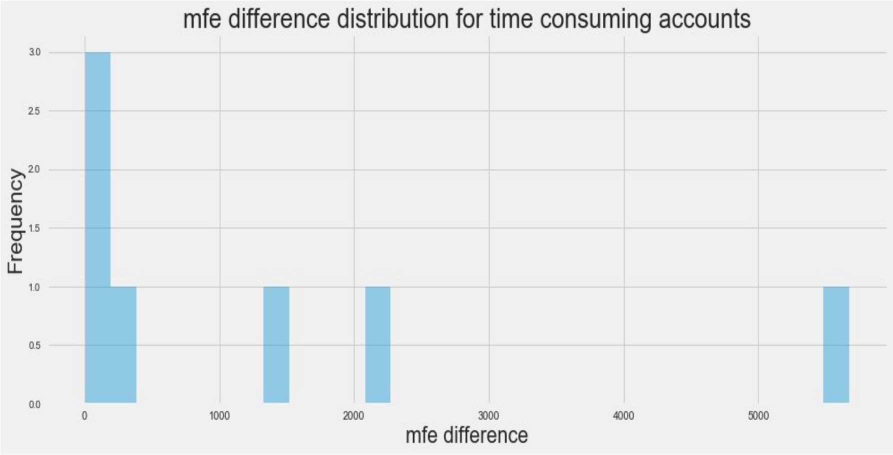
- Win distribution where Auto_ARIMA exceeds timeout threshold
    - time consuming accounts: 26/1000
    - time consuming accounts with Auto Arima as winning strategy: 14/26



Sixth Experiment: How to improve processing time?

- How much forecasting strength did we lose
  - Is ARIMA winning with huge margins

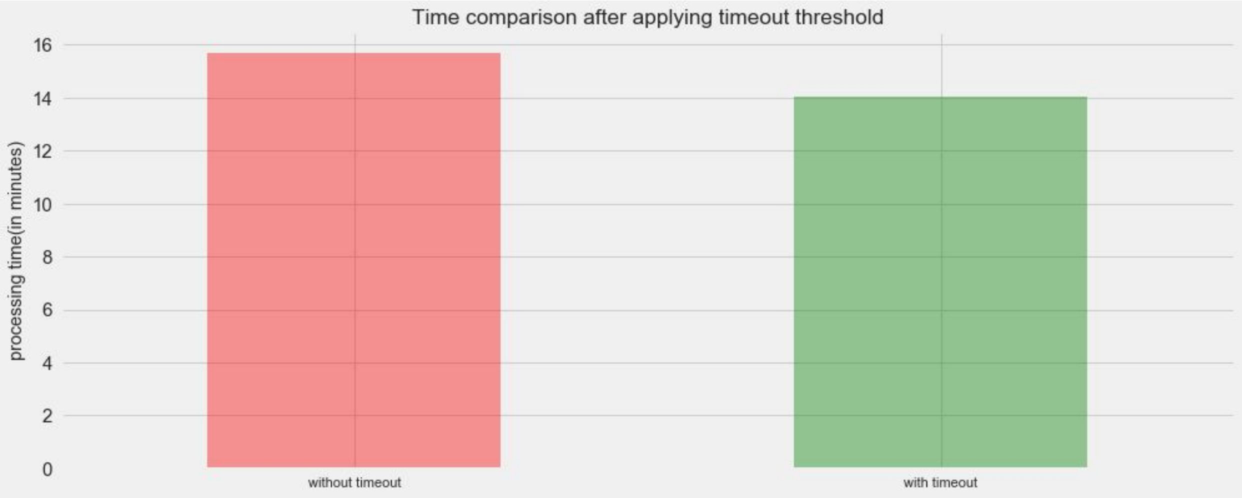




Sixth Experiment: How to improve processing time?

How does the time threshold help us:

- To gain 10% speedup
- Maintain acceptable loss of forecasting strength

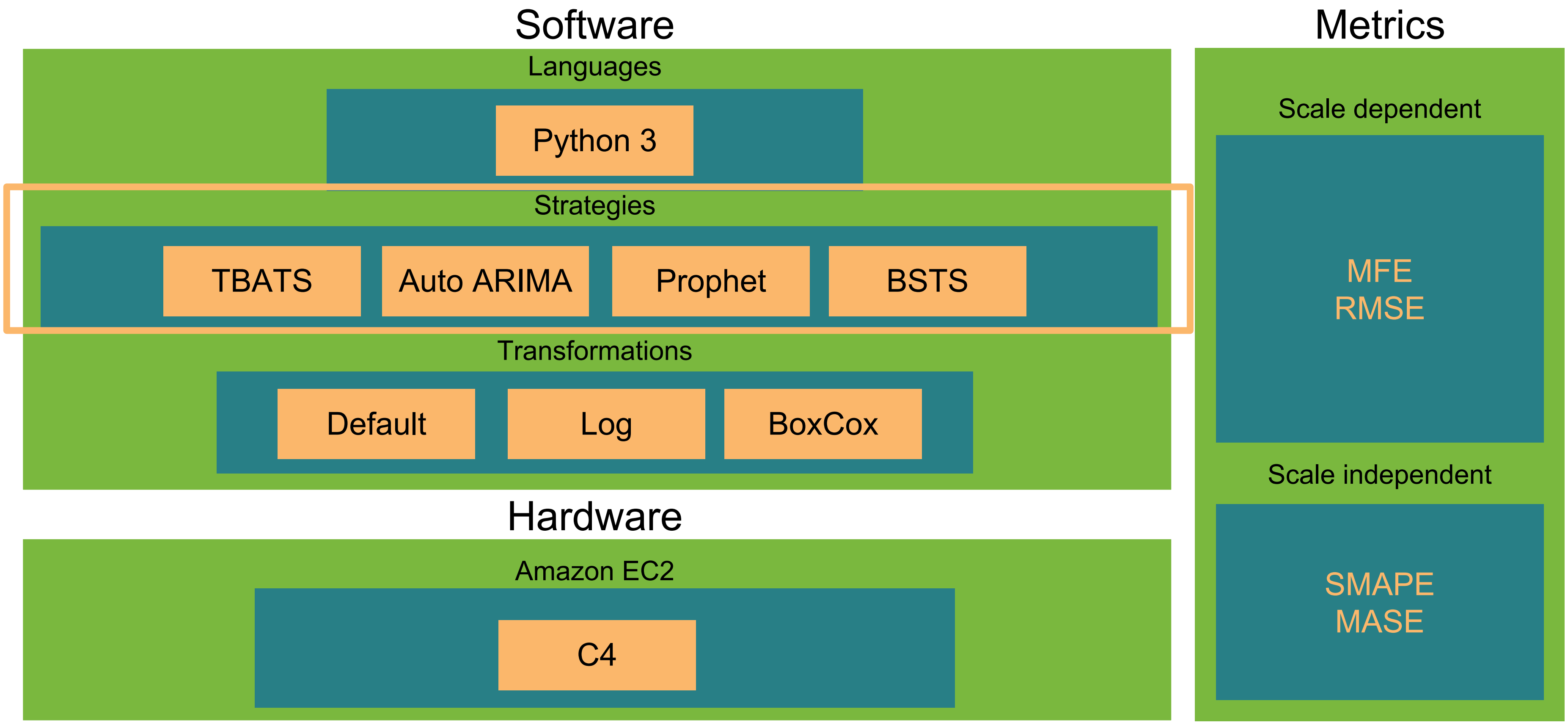


Sixth Experiment: How to improve processing time?

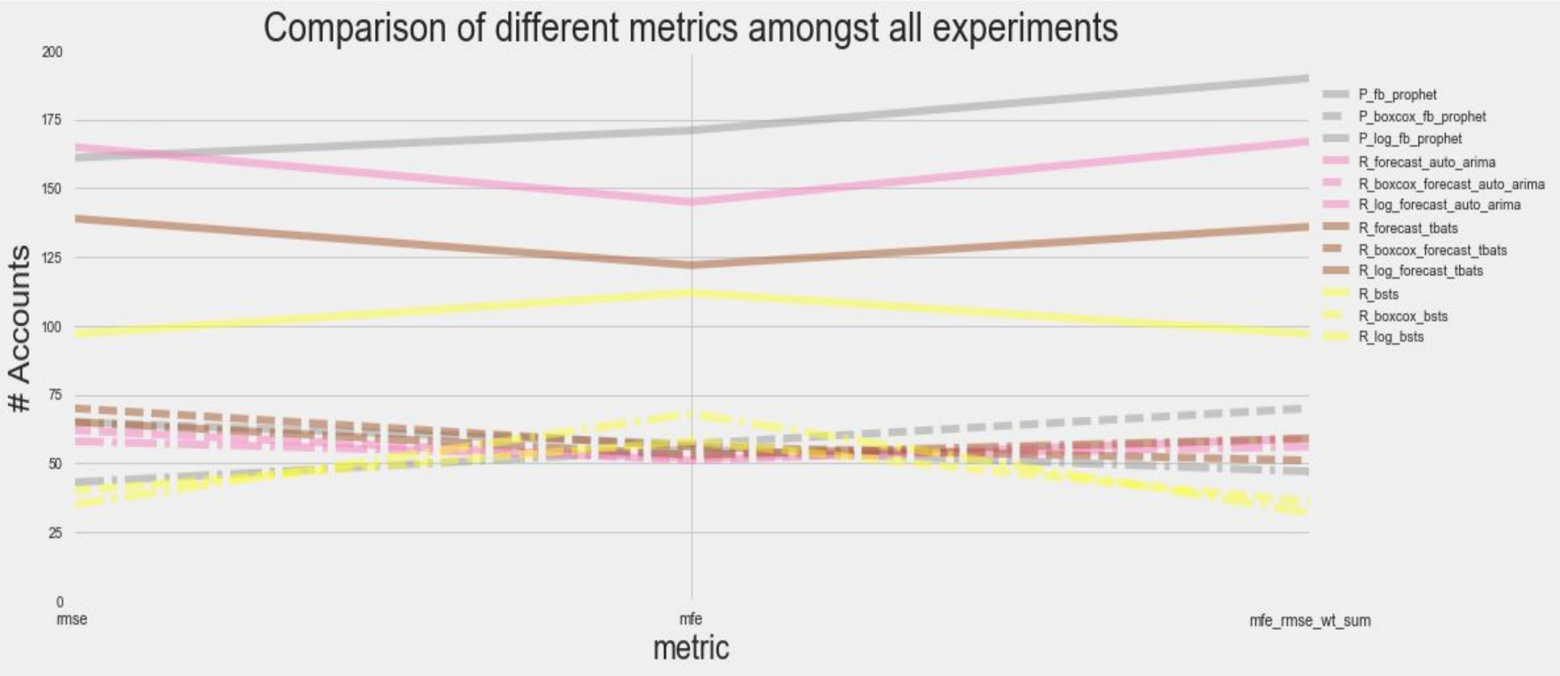
Rpy2 ineffectiveness: It is not feasible to timeout the process when using rpy2

Solution: Create individual R scripts for each R strategy and run them as a sub-process in Python

Add BSTS(Bayesian Structural Time Series) strategy



Add BSTS(Bayesian Structural Time Series) strategy



Concluding Pipeline

Software

Languages

- Python 3

Strategies

- TBATS
- Auto ARIMA
- Prophet
- BSTS

Transformations

- Default
- Log
- BoxCox

Hardware

Amazon EC2

- C4

Metrics

Scale dependent

- MFE
- RMSE

Scale independent

- SMAPE
- MASE

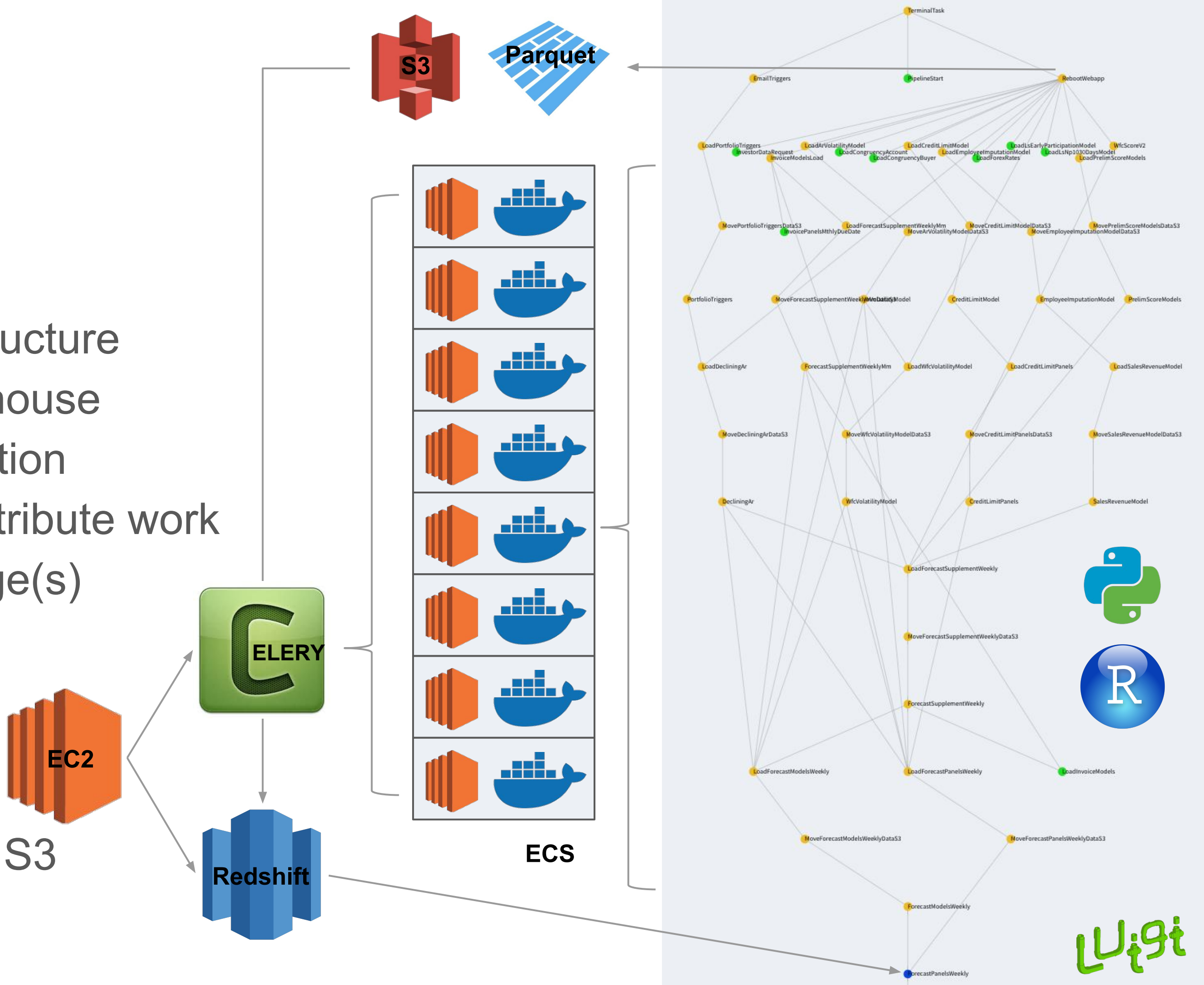
Scale + Cost Optimization

- Portable environments
- Immutable image
- Fast & Easy to deploy across a cluster
- Same code in production as in local development

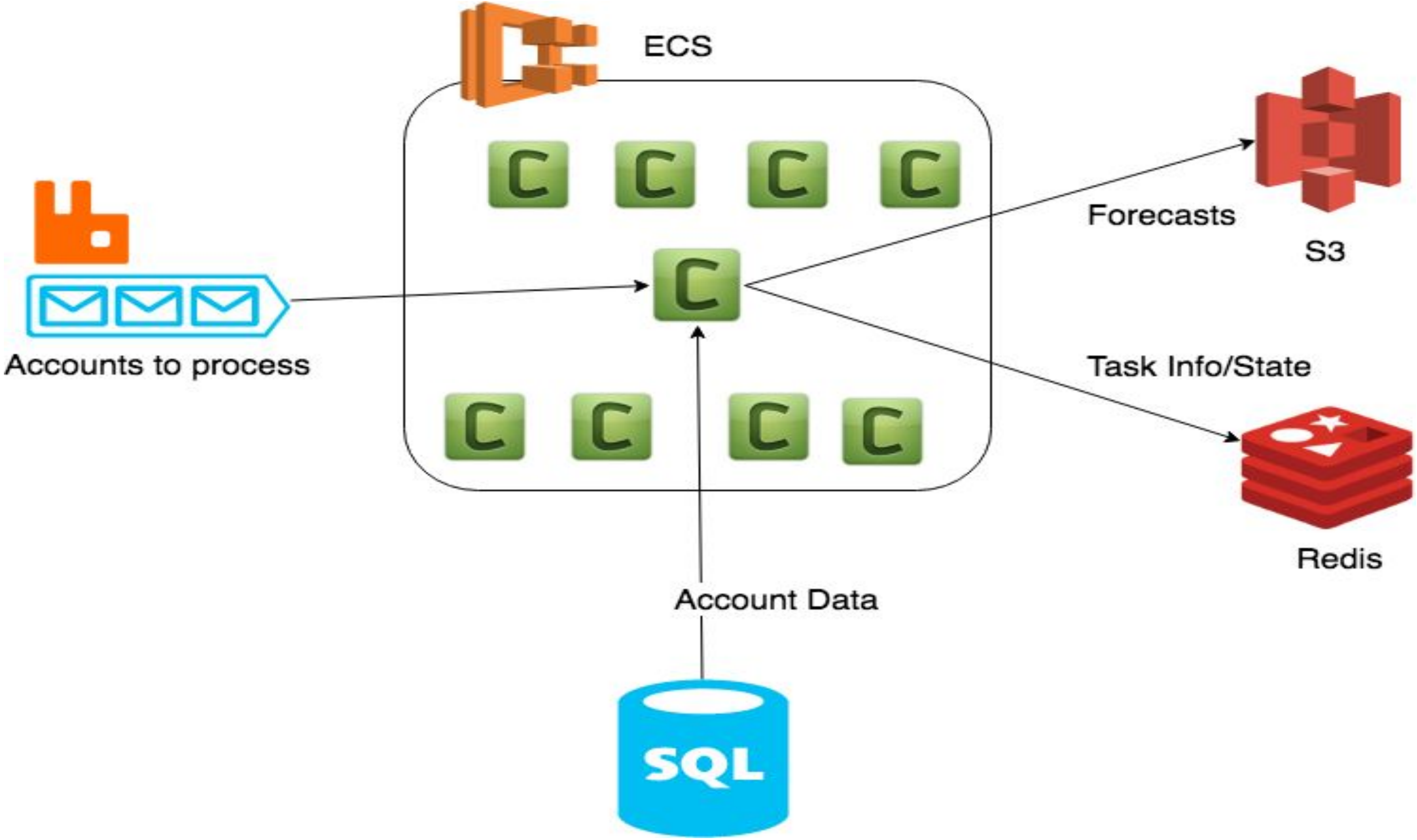
Architecture

The Basics:

- AWS elastic infrastructure
- Redshift data warehouse
- Luigi task orchestration
- Celery queue to distribute work
- Docker on c4.8xlarge(s)
- Amazon Linux OS
- Anaconda distro
  - Python 3.6
  - R 3.4
- Apache Parquet on S3



Concurrency Implementation

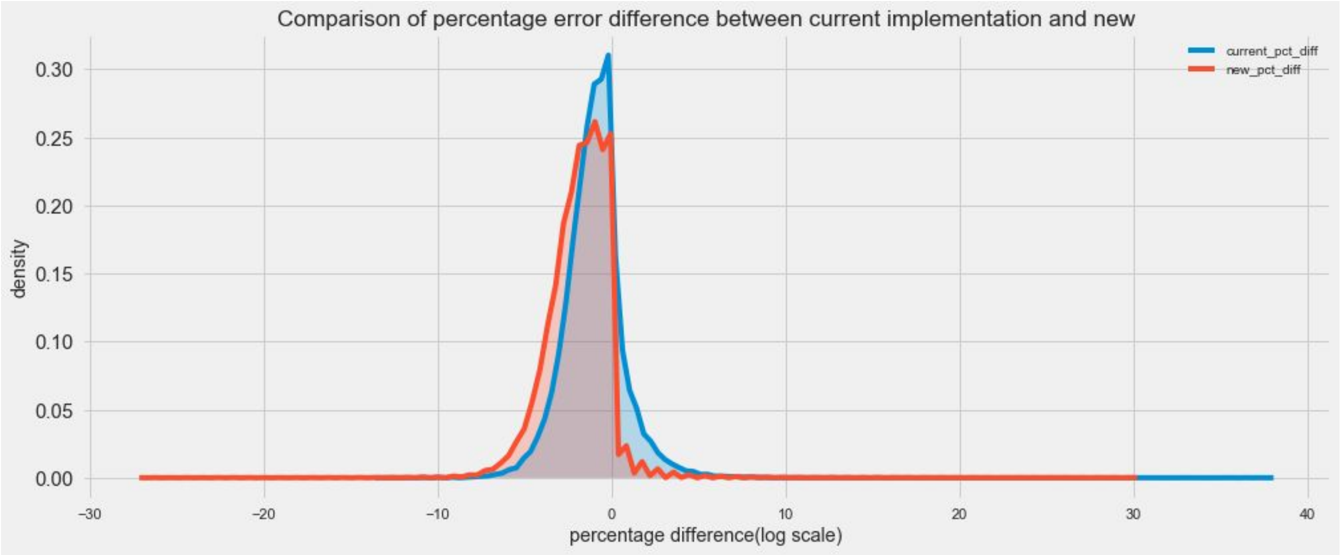


Cluster Stats

- 28 Instances (c4.8xlarge)
- 1000 workers
- ~2.6 Million forecasts
- ~5.5 hours

	On Demand	**Spot**
Cost per hour	$1.59	$0.60
Cost per run	$267.00	$100.80

Conclusion



New forecasting pipeline resulted in reduction of forecast error

Future Improvements

- Run the experiment if any of the library experience upgrades
- Periodically look for new time series strategies
- Add regression models
- Investigate AWS Batch or Kubernetes.

References

https://c2fo.com/

https://www.otexts.org/fpp

https://arxiv.org/pdf/1302.6613.pdf

https://sites.google.com/site/stevethebayesian/googlepageforstevenlscott/course-and-seminar-materials/bsts-bayesian-structural-time-series

Questions?

@pranav_bahl