# Travel Reimbursement

Hari Calzi

Business Information Systems
Prof. Paolo Ceravolo
a.a. 2024 - 2025

# Summary

# Description of the case study

This project aims to leverage process mining techniques to find some optimizations for the management of travel reimbursement process.

## Case study

The case study analyzed is related to the BPI Challenge 2020. Organizations commonly cover employee travel expenses for activities like client visits, conferences or project meetings. **Eindhoven University of Technology (TU/e)** follows a similar practice, with its staff frequently traveling for academic and professional reasons.

A key distinction is made between **domestic trips**, which require no prior approval and are reimbursed post-travel, and **international trips**, which instead require pre-approval through a travel permit. Regardless of the type of trip, employees must file a claim for reimbursement once expenses are incurred or within two months after the trip concludes.

## Dataset

The data are split into 5 different datasets, each one representing an event log:

1. **PermitLog**: it's related to travel permits, including all related events of prepaid travel cost declarations and travel declarations. It's composed of 6,886 cases, 36,796 events.
2. **InternationalDeclarations**: it includes requests related to international trips. It's composed of 6,449 cases, 72151 events.
3. **PrepaidTravelCost**: it tracks requests for pre-paid travel expenses where the associated service has not yet been used. It's composed of 10,500 cases, 56,437 events.
4. **DomesticDeclarations**: it includes requests related to national trips. It's composed of 10,500 cases, 56,437 events.
5. **RequestForPayment**: it's related to expense not associated with trips, like representation costs, hardware purchased for work. It's composed of 6,886 cases, 36,796 events.

Each event log is composed of a lot of different **attributes**. Here there's an explanation of the most important ones across all the logs:

- *id*: the id of the event

- *time:timestamp*: the timestamp in which the event occurred
- *case:id*: the id of the case related to an event
- *case:concept:name*: the name of the operation of the case
- *org:resource*: the resource which performs the operation, so the system or a staff member
- *org:role*: the role of the staff member that performs the operation

The datasets also include several financial attributes, such as *case:BudgetNumber*, *case:Amount*, *case:RequestedAmount*, *case:Permit RequestedBudget*, *case:AdjustedAmount*. The analysis proposed doesn't focus on these attributes, as initial investigations revealed a challenge in identifying precise correlations among them, coupled with their non-uniform distribution across the datasets, leading the focus of the analysis elsewhere.

After an in-depth study of the event logs and their interdependencies, it has been decided to consider each event log separately, at least in the first part of the analysis. This decision was primarily driven by two factors: firstly, DomesticDeclarations and RequestForPayment are not related at all with any other event log. Secondly, the other three logs, PermitLog, PrepaidTravelCost and InternationalDeclarations, share common attributes, but the relation between them are not easily understandable, especially for cardinality reasons. The possibility of joining them has been discarded to avoid an increase in complexity, even considering the current complexity of elogs.

# Organisational goals

As the case study is all about how travel and expense claims processes are done at TU/e, the organization's goals are not made explicit. Given the nature of these processes, it's possible to conclude that the organization's goals that this case study wants to support is to **enhance overall operational efficiency**.

This main goal can be divided into more subgoals, such as:

- reduce processes time
- minimize rework activities
- increase process conformance
- optimize the workload of each role
- minimize rejection rates

These goals are addressed in this report through the application of process mining. The methodology and key analytical steps are detailed in the Knowledge Uplift Trail section.

# Knowledge Uplift Trail

The **Knowledge Uplift Trail (KUT)** is a method used to turn raw data into useful insights. It involves a series of steps that eventually provide a clearer understanding of the underlying properties of the data, extracting knowledge from them. This is the KUT adopted in this project:

| Step | Input | Acquired Knowledge | | Output |
|---|---|---|---|---|
| | | Analytics / Model | Type | |
| 1 | Event logs | Data cleaning | Descriptive | Cleaned event logs |
| 2 | Cleaned event logs | Segmentation by role and resource | Descriptive | Segmented event logs |
| 3 | Segmented event logs | Comparative analysis: case duration, case size, reworked activities, rejection rates, conformance checking | Descriptive | Analytics results |
| 4 | Analytics results | Process improvement | Prescriptive | Optimization Opportunities |

Initially, each event log was cleaned to remove noise and irrelevant data, resulting in five purified event logs, ready to be used. Subsequently, the event logs were segmented at the case level, each one in the same way, to enable comparative analysis, producing five segmented event logs. These segmented logs were then analyzed in depth, allowing for the extraction of useful information and statistics. This analytics result was utilized in the final phase of the KUT to identify concrete optimization opportunities.

This KUT has been proposed in relation to the **research questions** proposed in the BPI challenge, in particular it was asked to:

- define a filtering step to remove noise or irrelevant data, achieved in step 1.
- Define segments of the log distinguishing cases, achieved in step 2.
- Compare the segments to verify their significant correlations with properties that may be connected to effectiveness, achieved in step 3.

- Assess whether the process minimizes complications and identify potential improvements to enhance efficiency, achieved in step 4.

# Step 1: data cleaning

Data cleaning step has been the most time-expensive activity, as there were five heterogeneous event logs. This section illustrates the actions performed to remove noise, irrelevant data and outliers from the logs. It's composed of two parts: in the first one the techniques used are illustrated, in the second one some considerations for each event log are made.

## Techniques used to remove noise and irrelevant data

Firstly, the event log has been checked to find **NaN values**, values that are usually returned as the result of an operation on invalid input operands:

```
# Check for NaN values across all columns
nan_counts = df.isnull().sum()

# Display only columns that have NaN values
print("Count of NaN values per column:")
print(nan_counts[nan_counts > 0])

if nan_counts.sum() == 0:
    print("\nNo NaN values found in the DataFrame.")
else:
    print(f"\nTotal NaN values found: {nan_counts.sum()}")
```

NaN were not the only strange values, in fact the attribute *case:Activity* had a lot of **UNKNOWN activities**. A function to evaluate this fact has been created:

```
# Get counts of each unique value in 'case:Activity'
activity_counts = df['case:Activity'].value_counts()

# Get percentages of each unique value in 'case:Activity'
activity_percentages = df['case:Activity'].value_counts(normalize=True) * 100

# Combine them
combined_activity_info = pd.DataFrame({
    'Count': activity_counts,
    'Percentage': activity_percentages
})

print("Counts and Percentages of each unique value in 'case:Activity':")
print(combined_activity_info)
```

In certain logs the attribute *org:role* presented a few **UNDEFINED roles**. To understand the entity of this problem a function has been implemented:

```
# Count the undefined roles
undefined_count = df['org:role'].astype(str).str.contains('UNDEFINED').sum()
```

Another strategy adopted was the **study of variants**, useful because they reveal the true diversity and complexity of how a process executes. Firstly, variants have been discovered through PM4PY:

```
variants_df = pm4py.get_variants_paths_duration(
    df,
    activity_key='concept:name',
    case_id_key='case:id',
    timestamp_key='time:timestamp'
)
```

Secondly, the **TOP N variants** have been evaluated to better understand the percentage of cases that they cover:

```
# Dataframe with sorting of the variants
variants = variants_df.groupby('@@variant_column').agg({'@@variant_count':
'first'}).reset_index()

# Sorting the DataFrame by '@@variant_count' column in descending order
variants = variants.sort_values(by='@@variant_count',
ascending=False).reset_index(drop=True)

# Display the top N variants
top_n = 5
print(top_n, " more frequent variants")
print(variants.head(top_n))

# Evaluate the coverage of the top N variants
total_cases = variants['@@variant_count'].sum()
cases_in_top_n = variants['@@variant_count'].head(top_n).sum()
percentage_coverage = (cases_in_top_n / total_cases) * 100
print(f"\nTop {top_n} variants cover {percentage_coverage:.2f}% of total cases")
```

In some cases, the dataframe has been filtered based on this top k variant, always using PM4PY, to remove the ones that lead to potential noise or very infrequent behavior:

```
filtered_df = pm4py.filter_variants_top_k(
    df,
    5,
    activity_key='concept:name',
    case_id_key='case:id',
    timestamp_key='time:timestamp'
)
```

If the variant analysis didn't retrieve useful information, it was possible to study the **start and end activities**, exploring very infrequent behaviors in the log and filtering out some cases related to these behaviors:

```
start_activities = pm4py.get_start_activities(
    filtered_df_dur,
    activity_key='concept:name',
    case_id_key='case:id',
    timestamp_key='time:timestamp'
)

filtered_df_dur = pm4py.filter_start_activities(
    filtered_df_dur,
    activities_to_exclude,
    retain=False,
    activity_key='concept:name',
    case_id_key='case:id',
    timestamp_key='time:timestamp'
)

end_activities = pm4py.get_end_activities(
    filtered_df_dur,
    activity_key='concept:name',
    case_id_key='case:id',
    timestamp_key='time:timestamp'
)

filtered_df_dur = pm4py.filter_end_activities(
    filtered_df_dur,
    activities_to_exclude,
    retain=False,
    activity_key='concept:name',
    case_id_key='case:id',
    timestamp_key='time:timestamp'
)
```

This analysis was helpful also to understand the behavior of the processes, understanding how they begin and end. The activity has been filtered also by their **frequency** in the log, filtering out the ones which appear in the log a very few times, compared to others.

```
# Compute frequency of activities
activity_counts =
filtered_df_dur["concept:name"].value_counts().sort_values(ascending=True)
print("Frequency of Activities:\n", activity_counts)
```

```
# Identify activities that are rare (frequency < 1000)
rare_activities_to_remove_cases = activity_counts[activity_counts <
1000].index.tolist()
print(f"Number of activities considered 'rare' (frequency < 1000):
{len(rare_activities_to_remove_cases)}")

# Find all 'case:id's that contain at least one of these rare activities
cases_with_rare_activities_events = filtered_df_dur[
    filtered_df_dur['concept:name'].isin(rare_activities_to_remove_cases)
]

# Extract the unique 'case:id's from these cases
case_ids_to_remove = cases_with_rare_activities_events['case:id'].unique()
print(f"\nNumber of cases that will be removed: {len(case_ids_to_remove)}")

# Filter the DataFrame to remove all rows corresponding to these cases, creating a new
DataFrame
filtered_df_cases_by_activities = filtered_df_dur[
    ~filtered_df_dur['case:id'].isin(case_ids_to_remove) # ~ negates the condition to
keep only those cases not in the list
].copy()

print(f"\nOriginal DataFrame rows: {len(filtered_df_dur)}")
print(f"Filtered DataFrame rows: {len(filtered_df_cases_by_activities)}")
print(f"Number of unique cases in original DataFrame:
{filtered_df_dur['case:id'].nunique()}")
print(f"Number of unique cases in filtered DataFrame:
{filtered_df_cases_by_activities['case:id'].nunique()}")
```

Another interesting technique adopted was to compute the **duration of each case** in the event log, through the *time:timestamp* attribute. The duration has been computed in days, hours, minutes, seconds and a check for NaN values has been implemented:

```
# Add a new DataFrame to store the durations of each case
filtered_df_dur = filtered_df.copy()

# Min and Max Timestamps for Case Durations
min_timestamps = filtered_df_dur.groupby('case:id')['time:timestamp'].min()
max_timestamps = filtered_df_dur.groupby('case:id')['time:timestamp'].max()

# Case Durations as Timedelta and convert to seconds
case_durations_timedelta = max_timestamps - min_timestamps
case_durations_seconds = case_durations_timedelta.dt.total_seconds()

# Mapping, conversion and new columns
filtered_df_dur.loc[:, 'case_duration_seconds'] =
filtered_df_dur['case:id'].map(case_durations_seconds)
```

```
filtered_df_dur.loc[:, 'case_duration_minutes'] =
filtered_df_dur['case_duration_seconds'] / 60
filtered_df_dur.loc[:, 'case_duration_hours'] =
filtered_df_dur['case_duration_seconds'] / 3600
filtered_df_dur.loc[:, 'case_duration_days'] = filtered_df_dur['case_duration_seconds']
/ (24 * 3600)

# Check for Nan
print(f"N° of NaN created: {filtered_df_dur['case_duration_seconds'].isna().sum()}")
```

This filtered dataframe with duration has been used to check if any cases had **zero duration**, leading to potential errors in the log:

```
# Find the number of cases with zero duration
zero_duration_cases_count = (filtered_df_dur['case_duration_seconds'] == 0).sum()

print(f"Number of cases with zero duration (seconds): {zero_duration_cases_count}")
```

Finally, to better understand if the data cleaning procedure had reduced the amount of noise, errors and outliers, a **Directly Follows Graph (DFG)** and a **Heuristic Miner (HM)** map have been computed:

```
# Directly Follows Graph (DFG)
# Create graph from original DF and visualise it
dfg, start_activities, end_activities = pm4py.discover_dfg(df)
pm4py.view_dfg(dfg, start_activities, end_activities)

# Create graph from filtered DF and visualise it
dfg, start_activities, end_activities = pm4py.discover_dfg(df_cleaned)
pm4py.view_dfg(dfg, start_activities, end_activities)

# Heuristic Miner (HM)
# Discover the HM-map of the original DF and visualise it
map = pm4py.discover_heuristics_net(df)
pm4py.view_heuristics_net(map)

# Discover the HM-map of the filtered DF and visualise it
map = pm4py.discover_heuristics_net(df_cleaned)
pm4py.view_heuristics_net(map)
```

### Considerations for each event log

In this section some considerations about each event log, the filtering technique adopted in it and the result obtained are presented. For readability reasons, only the most valuable considerations are reported.

In the **RequestForPayment** event log, during the variant analysis, the top 5 variants covered 88.63% of total cases. This meant that very few behaviors covered most of cases, so the event log has been filtered based on this assumption. Luckily there were no NaN values or zero duration events.

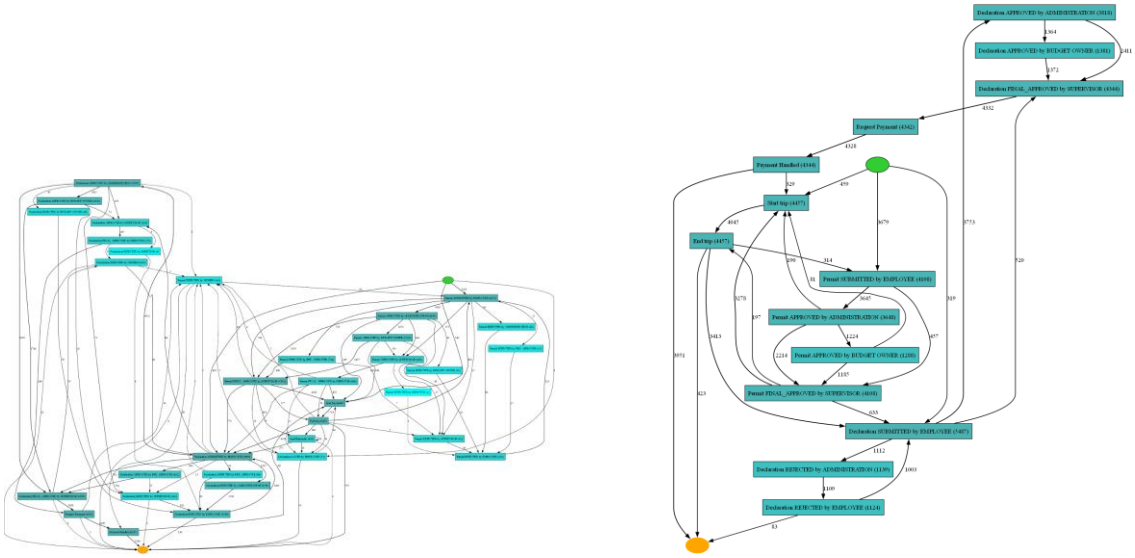
The HM map presented here visually demonstrates the simplification of the log's structure, achieved by removing errors, noise and rare process deviations.


This preparatory step was instrumental in ensuring a more focused and easier analysis in the subsequent stages of the KUT.

In the **DomesticDeclarations** log the situation was very similar to the previous one, in fact the top 5 variants covered 90.71% of total cases, no errors were found and the visualizations tools provided confirmation of the work done.

In the **InternationalDeclarations** log the situation was a slightly different because there was a column, *case:Permit ActivityNumber*, which had to be removed as 93.28% of its values were UNKNOWN. Variant wasn't useful here, in fact the top 10 variants covered

only 55.11% of cases, meaning a less standardized behavior of the processes. In this case a lot of activities with a very infrequent behavior were filtered out, either in start, "central" or end activities. This leads to a clearer event log, much more understandable with respect to the original one but also with some deviations, as shown in the HM maps.



In the **PrepaidTravelCost** log the situation of the various statistics computed to understand what needed to be filtered out, especially of the top k variant, was in the middle between the previous logs. Specifically, the top 10 variants accounted for 74% of all cases. After careful consideration, a decision was made to filter the log based on these dominant variants. While excluding 26% of the cases might appear substantial, this approach proved highly beneficial for obtaining a cleaner and more focused log, which significantly enhanced the utility of subsequent analyses.

The last log, **PermitLog**, was quite difficult to manage as it was the least linear and more complex among all. Firstly, it had 153 attributes with more than 50% of NaN values. As these attributes were not at the center of the goals of the analysis, they had been removed through this function:

```
# Count NaN values in each column and calculate their percentage
nan_counts = df.isnull().sum()
total_rows = len(df)
nan_percentages = (nan_counts / total_rows) * 100
print("NaN percentage for each column:\n")
print(nan_percentages)
```

```
# Identify columns with at least 50% NaN values
columns_with_high_nan = nan_percentages[nan_percentages >= 50]
print("\nColumns with at least 50% NaN values:", len(columns_with_high_nan))

# Drop those columns from the DataFrame
columns_to_drop = columns_with_high_nan.index.tolist()
filtered_df = df.drop(columns=columns_to_drop)
```

Variants here were not useful as the top 50 variants covered only 63.75% of total cases, so the cases were filtered based on frequency of activities. Graphically, the HM map shows as situation like the one obtained in *InternationalDeclarations* and *PrepaidTravelCost*.

The filtering procedures applied at the five logs produced five cleaned logs, each one exported through the functions **convert_to_event_log** and **write_xes** of PM4PY and ready to be segmented.

# Step 2: segmentation

The second step of the KUT was the segmentation phase, which aimed to define segments of the log distinguishing cases. As said before, the log wasn't joined, so this phase was crucial as it enabled the possibility of confrontation among them. This phase was also very important as good segmentation will lead to better analysis of the further phases of the KUT.

The key idea was to find some attributes that were common among each log and which can create segmentations useful for the analysis. After a deep phase of understanding of each log and its attributes, two potential candidates were found: the couple **org:resource** and **org:role**. These attributes, consistently present across all logs, identified the roles involved in the processes.

To evaluate this choice, the frequency of these attributes was computed:

```
resource_counts = df['org:resource'].value_counts()
print(resource_counts)

role_counts = df['org:role'].value_counts()
print("\n",role_counts)
```

After repeating this procedure in each event log, a correlation between them has been found: *org:resource* is a generalization of *org:role*. In fact, there are two types of resources: SYSTEM and STAFF MEMBER. Each time the resource is SYSTEM the role is set as UNDEFINED, as the system is not a STAFF MEMBER. Instead, when the resource is STAFF MEMBER there can be six different roles: EMPLOYEE, SUPERVISOR, ADMINISTRATION, BUDGET OWNER, DIRECTOR, PRE_APPROVER.

Segmenting the logs based on this attribute allowed for a granular understanding of each role's specific contributions, highlighting associated performance issues and revealing potential optimization opportunities.

To segment based on these roles, a custom function has been proposed:

```
# Map of roles to numeric values
role_to_numeric_map = {
    'UNDEFINED': 0,
    'EMPLOYEE': 1,
    'SUPERVISOR': 2,
    'ADMINISTRATION': 3,
    'BUDGET OWNER': 4,
```

```
    'DIRECTOR': 5,
    'PRE_APPROVER': 6,
}

# Function to create a numeric role column in the DataFrame
def create_numeric_role_column(df):

    # Work on a copy of the DataFrame to avoid problems
    df_copy = df.copy()

    # Default value for org_role_numeric
    df_copy.loc[:, 'org_role_numeric'] = -1

    # Map the roles to numeric values
    df_copy.loc[df_copy['org:role'].isin(role_to_numeric_map.keys()),
'org_role_numeric'] = df_copy['org:role'].map(role_to_numeric_map)

    return df_copy
```

Basically, it maps each role, also UNDEFINED which refers to SYSTEM, to a numeric value. Then it creates a new column in the dataframe with numeric values for each role, allowing for easier analysis based on this segmentation.

This operation was repeated for each log, then it was checked if any -1 values were left, meaning a bad mapping during the procedure, and the segmented datasets were exported, ready to be analyzed.

# Step 3: comparative analysis

The comparative analysis phase took as input the segmented datasets and produced as output analytics results. These analyses were made based on four different properties: case duration, case size, reworked activities, rejection rates, along with conformance checking.

To automate the analysis process among all the logs, a dictionary with the names of all the logs was created, to facilitate iterations.

```
# Store dataframe in a dictionary for easier iteration
all_dfs = {
    "PrepaidTravelCost": df_PTC,
    "RequestForPayments": df_RFP,
    "InternationalDeclarations": df_ID,
    "PermitLog": df_PL,
    "DomesticDeclarations": df_DD
}
```

Given that the segmentation was performed by role, for each property key statistics regarding cases in which each role was involved have been computed.

The next sections provide details about the methods adopted.

## Case duration

Case duration, a fundamental property, quantifies the total time required for a case to complete. It directly measures process efficiency and speed, indicating how quickly a case moves from start to finish. This metric is essential for identifying bottlenecks, excessive waiting times and areas where process flow can be significantly accelerated.

This is the custom function developed, the comments in the code are explicative of what it does:

```
# List to store summary data for easier comparison
summary_data = []

# Iteration through each DataFrame and perform analysis
for log_name, df in all_dfs.items():

    # Calculate overall log duration metrics
    overall_mean_duration = df['case_duration_days'].mean()
    overall_median_duration = df['case_duration_days'].median()

    # Group by the numeric role and calculate metrics for each segment
```

```
    segment_duration_stats =
df.groupby('org_role_numeric')['case_duration_days'].agg(['mean', 'median',
'count']).reset_index()

    # Add role names for better readability
    segment_duration_stats['org_role_name'] =
segment_duration_stats['org_role_numeric'].map(numeric_to_role_map)

    # Store summary data for overall comparison
    for _, row in segment_duration_stats.iterrows():
        summary_data.append({
            'Log': log_name,
            'Role_Numeric': row['org_role_numeric'],
            'Role_Name': row['org_role_name'],
            'Mean_Duration_Days': row['mean'],
            'Median_Duration_Days': row['median'],
            'Case_Count_in_Segment': row['count']
        })

# Overall Comparison Table, creating a DataFrame from the summary data
overall_duration_summary_df = pd.DataFrame(summary_data)
print("\nOverall Duration Summary Across All Logs and Segments")

# Sort by median duration and display
print(overall_duration_summary_df.sort_values(by='Median_Duration_Days',
ascending=False).to_string())

# Group by Role Name across all logs, aggregating the metrics
print("\n\nStatistics per role across all Logs:\n")

role_stats = overall_duration_summary_df.groupby('Role_Name').agg(
    Avg_of_Median_Duration_Days=('Median_Duration_Days', 'mean'),
    Avg_of_Mean_Duration_Days=('Mean_Duration_Days', 'mean'),
    Total_Cases_Associated=('Case_Count_in_Segment', 'sum')
).reset_index()

# Sort for easier interpretation and display
role_stats = role_stats.sort_values(by='Avg_of_Median_Duration_Days', ascending=False)
print(role_stats.to_string(index=False))
```

## Case size

Case size represents the total volume of activities related to a single case from start to finish. It's crucial as it directly quantifies the complexity and length of individual process instances, often indicating potential inefficiencies.

```
# List to store summary data for easier comparison
summary_data_case_size = []
```

```
# Iteration through each DataFrame and perform analysis
for log_name, df in all_dfs.items():

    # Calculate case size for each case, merging the size back into the DataFrame
    case_sizes_series = df.groupby('case:id').size().rename('case_size')
    df_with_case_size = df.merge(case_sizes_series.reset_index(), on='case:id',
how='left')

    # Calculate overall log case size metrics
    unique_case_sizes_for_overall =
df_with_case_size.drop_duplicates(subset=['case:id'])['case_size']
    overall_mean_case_size = unique_case_sizes_for_overall.mean()
    overall_median_case_size = unique_case_sizes_for_overall.median()

    # Group by the numeric role and calculate case size metrics for each segment.
    segment_case_size_stats =
df_with_case_size.groupby('org_role_numeric')['case_size'].agg(
        mean='mean',
        median='median',
        count='count'
    ).reset_index()

    # Add role names
    segment_case_size_stats['org_role_name'] =
segment_case_size_stats['org_role_numeric'].map(numeric_to_role_map)

    # Store summary data for overall comparison
    for _, row in segment_case_size_stats.iterrows():
        summary_data_case_size.append({
            'Log': log_name,
            'Role_Numeric': row['org_role_numeric'],
            'Role_Name': row['org_role_name'],
            'Mean_Case_Size': row['mean'],
            'Median_Case_Size': row['median'],
            'Event_Count_in_Segment': row['count']
        })

# Overall Comparison Table, creating a DataFrame from the summary data
overall_case_size_summary_df = pd.DataFrame(summary_data_case_size)
print("\nOverall Case Size Summary Across All Logs and Segments")

# Sort by median case size and display
print(overall_case_size_summary_df.sort_values(by='Median_Case_Size',
ascending=False).to_string())

# Group by Role Name across all logs, aggregating the metrics
print("\n\nStatistics per Role across all logs:\n")
```

```
consolidated_role_size_stats = overall_case_size_summary_df.groupby('Role_Name').agg(
    Avg_of_Median_Case_Size=('Median_Case_Size', 'mean'),
    Avg_of_Mean_Case_Size=('Mean_Case_Size', 'mean'),
    Total_Events_Associated=('Event_Count_in_Segment', 'sum')
).reset_index()

# Sort for easier interpretation and display
consolidated_role_size_stats =
consolidated_role_size_stats.sort_values(by='Avg_of_Median_Case_Size', ascending=False)
print(consolidated_role_size_stats.to_string(index=False))
```

## Reworked activities

Reworked activities refer to instances where tasks or entire process segments are repeated, often due to errors, incomplete information or rejections.

```
# List to store summary data
summary_data_rework = []

# Iteration through each DataFrame
for log_name, df in all_dfs.items():

    # Get the rework cases per activity
    rework_per_activity = pm4py.stats.get_rework_cases_per_activity(
        df,
        activity_key='concept:name',
        case_id_key='case:id',
        timestamp_key='time:timestamp'
    )

    # Convert the rework cases dictionary to a DataFrame
    rework_df = pd.DataFrame.from_dict(rework_per_activity, orient='index',
columns=['cases_with_rework_for_activity']).reset_index()
    rework_df.rename(columns={'index': 'concept:name'}, inplace=True)
    activity_role_mapping = df[['concept:name',
'org_role_numeric']].drop_duplicates().dropna(subset=['org_role_numeric'])
    rework_by_activity_and_role = rework_df.merge(activity_role_mapping,
on='concept:name', how='left')

    # Compute rework statistics per role
    rework_stats_per_role =
rework_by_activity_and_role.groupby('org_role_numeric').agg(
        total_cases_with_rework_activities=('cases_with_rework_for_activity', 'sum')
    ).reset_index()
    total_cases_involved_per_role =
df.groupby('org_role_numeric')['case:id'].nunique().reset_index(name='total_cases_invol
ved')
    rework_stats_per_role = rework_stats_per_role.merge(total_cases_involved_per_role,
on='org_role_numeric', how='left').fillna(0)
```

```
    rework_stats_per_role['percentage_cases_with_rework_by_role'] =
(rework_stats_per_role['total_cases_with_rework_activities'] /
rework_stats_per_role['total_cases_involved']) * 100
    rework_stats_per_role['Role_Name'] =
rework_stats_per_role['org_role_numeric'].map(numeric_to_role_map)

    # Iterate through the rework statistics and append to summary data
    for _, row in rework_stats_per_role.iterrows():
        summary_data_rework.append({
            'Log': log_name,
            'Role_Numeric': row['org_role_numeric'],
            'Role_Name': row['Role_Name'],
            'Total_Cases_with_Rework_Activities_for_Role':
row['total_cases_with_rework_activities'],
            'Total_Cases_Involved_for_Role': row['total_cases_involved'],
            'Percentage_Cases_with_Rework_by_Role':
row['percentage_cases_with_rework_by_role']
        })

overall_rework_summary_df = pd.DataFrame(summary_data_rework)
print("Rework statistics per role across all logs:\n")

# Aggregate the rework statistics across all logs
consolidated_rework_stats = overall_rework_summary_df.groupby('Role_Name').agg(
    Avg_Percentage_Cases_with_Rework_by_Role=('Percentage_Cases_with_Rework_by_Role',
'mean'),
    Total_Rework_Cases_Associated=('Total_Cases_with_Rework_Activities_for_Role',
'sum'),
    Total_Cases_Involved_Overall=('Total_Cases_Involved_for_Role', 'sum')
).reset_index()

# Sort and display
consolidated_rework_stats =
consolidated_rework_stats.sort_values(by='Avg_Percentage_Cases_with_Rework_by_Role',
ascending=False)
print(consolidated_rework_stats.to_string(index=False))
```

## Conformance checking

In conformance checking, the assessment involves evaluating the alignment between actual process executions, derived from event logs, and a predefined process model. As a reference model was not available, a Petri net was automatically discovered from each event log using the Inductive Miner algorithm. This discovered Petri net then served as the model for comparison against the real behavior captured within its respective log, enabling the calculation of fitness and precision scores.

Fitness measures how much of the behavior observed in the event log is explained by the process model, indicating if the model allows for all the real process executions. Precision, conversely, assesses how much behavior allowed by the model actually appears in the event log, indicating if the model is too permissive or "underfitting" the reality.

```
summary_conformance_data = []

for log_name, df in all_dfs.items():
    print(f"\nAnalyzing Log: {log_name}")

    # Convert DataFrame to Event Log
    log = pm4py.convert_to_event_log(df, case_id_key='case:id',
activity_key='concept:name', timestamp_key='time:timestamp')

    # Discover the Petri net using Inductive Miner
    net, im, fm = discover_petri_net_inductive(log)

    # Fitness computation using fitness_token_based_replay
    fitness_score = None
    fitness_results = fitness_token_based_replay(log, net, im, fm)
    if isinstance(fitness_results, dict) and 'log_fitness' in fitness_results: # Avoid
potential KeyError
        fitness_score = fitness_results['log_fitness']

    # Precision computation using precision_token_based_replay
    precision_score = None
    precision_score = precision_token_based_replay(log, net, im, fm)
    if isinstance(precision_score, dict) and 'precision' in precision_score: # Avoid
potential KeyError
        precision_score = precision_score['precision']

    # Summary
    summary_conformance_data.append({
        'Log': log_name,
        'Fitness': fitness_score,
        'Precision': precision_score
    })

# DataFrame for overall conformance summary, displaying fitness and precision scores
overall_conformance_df = pd.DataFrame(summary_conformance_data)
overall_conformance_df = overall_conformance_df.sort_values(by='Precision',
ascending=False)
print("\nOverall conformance Summary Across all logs:\n")
print(overall_conformance_df.to_string(index=False))
```

## Rejection rates

Rejection rates measure the frequency at which cases are refused at specific points within a process. High rejection rates often signal inefficiencies, errors in previous steps or issues with initial data quality, leading to increased rework.

```
rejected_cases_summary_list = []

# Iteration through each Dataframe
for log_name, df_log in all_dfs.items():

    # Preprocess the DataFrame to ensure correct types
    df_log_processed = df_log.copy()
    df_log_processed['concept:name'] = df_log_processed['concept:name'].astype(str)
    df_log_processed['org_role_numeric'] =
pd.to_numeric(df_log_processed['org_role_numeric'], errors='coerce').fillna(-
1).astype(int)

    # Filter the DataFrame for activities that contain 'REJECTED'
    rejected_activities_in_log =
df_log_processed[df_log_processed['concept:name'].str.contains('REJECTED', na=False)]

    # If there are rejected activities in the log
    if not rejected_activities_in_log.empty:
        # Calculate some statistics
        unique_rejected_cases = rejected_activities_in_log['case:id'].nunique()
        total_cases_in_log = df_log_processed['case:id'].nunique()
        percentage_cases_with_rejection = (unique_rejected_cases / total_cases_in_log)
* 100
        involved_roles_numeric =
rejected_activities_in_log['org_role_numeric'].unique()
        involved_roles_names = [numeric_to_role_map.get(role_num, 'UNKNOWN_ROLE') for
role_num in involved_roles_numeric]
        involved_roles_str = ', '.join(sorted(set(involved_roles_names)))
        total_rejected_activities_count = len(rejected_activities_in_log)

        # Append the summary for this log
        rejected_cases_summary_list.append({
            'Log_Name': log_name,
            'Unique_Cases_with_Rejection': unique_rejected_cases,
            'Percentage_Cases_with_Rejection_in_Log': percentage_cases_with_rejection,
            'Total_Rejected_Activities': total_rejected_activities_count,
            'Involved_Roles': involved_roles_str
        })
    # If there are no rejected activities in the log
    else:
        # Append a summary indicating no rejections found
        rejected_cases_summary_list.append({
            'Log_Name': log_name,
```

```
            'Unique_Cases_with_Rejection': 0,
            'Percentage_Cases_with_Rejection_in_Log': 0,
            'Total_Rejected_Activities': 0,
            'Involved_Roles': 'No Rejections Found'
        })

# Create a DataFrame from the summary list, order it by percentage of cases with
rejection and display it
final_consolidated_rejections_df = pd.DataFrame(rejected_cases_summary_list)
final_consolidated_rejections_df = final_consolidated_rejections_df.sort_values(
    by='Percentage_Cases_with_Rejection_in_Log', ascending=False
)

print("Overall rejection summary across all logs:\n")
print(final_consolidated_rejections_df.to_string(index=False))
```

## Results of the analysis

| Role name | Average of median duration in days | Average of mean duration in days | Total cases associated |
|---|---|---|---|
| Director | 38.30 | 59.58 | 226 |
| Budget owner | 34.79 | 43.16 | 10268 |
| Undefined | 33.81 | 44.63 | 52937 |
| Administration | 33.14 | 42.04 | 33523 |
| Employee | 33.95 | 43.11 | 60200 |
| Supervisor | 32.35 | 42.48 | 35861 |
| Pre approver | 14.63 | 20.53 | 1095 |

The analysis of **case duration** across all logs revealed distinct performance patterns across different roles. The focus was on the average median duration in days for robustness against outliers. The PRE_APPROVER role consistently stood out with a lower average median duration of 14.63 days, indicating its involvement in the quickest processing paths, and it's associated with a relatively small number of cases (1,095). The DIRECTOR and BUDGET OWNER roles were associated with the longest average median durations, 38.30 days and 34.79 days respectively. While the Director was involved in a small number of cases (226), the Budget Owner was associated with a substantial volume of cases (10,268), suggesting that their involvement might be a significant factor in the overall duration for a large portion of the process instances.

| Role name | Average of median case size | Average of mean case size | Total case associated |
|---|---|---|---|
| Budget owner | 9.2 | 9.45 | 10268 |
| Director | 9.0 | 9.88 | 226 |
| Administration | 8.4 | 8.86 | 33523 |
| Employee | 8.0 | 8.46 | 60200 |
| Undefined | 8.0 | 8.44 | 52937 |
| Supervisor | 7.8 | 8.43 | 35861 |
| Pre approver | 6.0 | 6.00 | 1095 |

Analyzing the average median **case size** provided insights into the complexity or length of process instances associated with each role. The BUDGET OWNER and DIRECTOR roles are linked to cases with the highest average median case sizes, 9.2 and 9.0 activities respectively. While the Director was involved in a small number of cases (226), the Budget Owner was associated with lots of cases (10,26). This association with larger cases reflected also their longer average median durations, indicating that more complex cases naturally take more time when these roles are involved. The PRE_APPROVER role consistently appearead in the simplest cases, showing an average median case size of 6.0 activities, mirroring its efficiency observed in case duration.

| Role name | Average percentage of cases with rework | Total rework cases associated | Total case involved |
|---|---|---|---|
| Undefined | 38.71 | 1792 | 4629 |
| Employee | 13.18 | 2520 | 20241 |
| Administration | 5.75 | 484 | 8275 |
| Supervisor | 3.99 | 193 | 4833 |
| Budget owner | 1.54 | 22 | 1424 |

Looking at the **rework statistics**, it was possible to see how often cases involve repeated steps for different roles. The Undefined role stood out significantly, with nearly 38.71% of its associated cases experiencing rework. This role refers to the system, so its high associated rework rate strongly suggested that there are significant opportunities for

improved automation or better system design to reduce inefficiencies. The 'Employee' role also showed a notable rework rate at 13.18%, affecting many cases (2,520 rework cases out of 20,241 total).

| Log | Fitness | Precision |
|---|---|---|
| RequestForPayments | 1.0 | 1.00 |
| DomesticDeclarations | 1.0 | 0.98 |
| InternationalDeclaration | 1.0 | 0.79 |
| DomesticDeclarations | 1.0 | 0.63 |
| PermitLog | 1.0 | 0.27 |

The **conformance checking** analysis showed how well the discovered models, using Inductive Miner, fit the actual process data. For all logs, the fitness score was consistently 1.0, meaning that the model created could perfectly explain every action found in the real event logs. However, precision scores varied significantly. A high precision (like RequestForPayments at 1.00 and DomesticDeclarations at 0.98) meant the model only allowed behaviors that were seen in the log, so it was a very good fit. But for other logs, especially PermitLog, the precision was lower (0.27). This indicated that the model was too general, allowing many process paths that were not actually observed in real data. This suggested these processes might be less structured or have many unique variations not captured efficiently by a simple model.

| Log | Unique cases with rejections | Percentage of cases with rejection | Total rejected activities | Involved roles |
|---|---|---|---|---|
| InternationalDeclaration | 957 | 21.47 | 2263 | Administration, employee |
| PermitLog | 818 | 16.93 | 2006 | Administration, employee |
| RequestForPayments | 343 | 5.68 | 686 | Administration, employee |

| | | | | |
|---|---|---|---|---|
| DomesticDeclarations | 345 | 3.67 | 690 | Administration, employee |
| DomesticDeclarations | 53 | 3.42 | 106 | Administration, employee |

The analysis of **rejection rates** revealed clear differences across the various logs. The International Declarations log showed the highest rate, with over 21% of cases experiencing a rejection, accounting for a significant number of rejected activities (2,263). In contrast, RequestForPayments (5.68%), DomesticDeclarations (3.67%), and PrepaidTravelCost (3.42%) showed much lower rejection percentages. Interestingly, the roles of Administration and Employee were involved in all rejected cases across every log, while other roles weren't. This sequence, where an administrative rejection was always followed by a rejection associated with the employee as shown in HM maps and DFGs, suggested the latter activity may primarily function as an automated notification step rather than an independent process action.

It's important to note that these observations were based on the filtered event logs, implying that any rejection cases linked to very infrequent or outlier behaviors (and potentially other roles) have been excluded during the data cleaning phase.

# Step 4: process improvement

This section leverages the insights gained from the process mining analysis to evaluate the current state of the travel and expense reimbursement processes and to propose concrete improvement opportunities.

## Assessment of process complications

Based on the detailed analysis of event logs, it can be concluded that the current travel and expense reimbursement process does not effectively minimize complications, presenting several areas of inefficiency, unnecessary delays and sometimes high rejection rates.

The analysis revealed that the Director (with an average median duration of 38.30 days) and Budget Owner (34.79 days) roles are **bottlenecks**. Cases involving these roles required the longest processing times and are also the most complex (average of 9.0 and 9.2 activities). This suggested that their approval or review steps often involve significant **delays** or **very intensive processing**.

High rejection rates highlighted process complications. Over 21% of International Declarations and nearly 17% of Permit Logs were rejected, indicating **issues with initial submission quality or subsequent validation**. This directly led to **unnecessary rework**, wasting time and resources for both the Employee, who resubmits, and Administration, who re-processes.

With a 38.71% rework rate, the Undefined role (system activities) pointed to inefficiencies and **unnecessary back-and-forth** in automated parts of the process, not related to human activities.

Conformance checking results, especially for PermitLog (0.27 precision), highlighted a significant **lack of process standardization**. This low score meant the process model allowed for many behaviors not actually seen in practice, indicating high variability and numerous alternative ways of working. This made the process difficult to manage and to predict.

## Potential Improvements and Optimization Opportunities

Several opportunities for process improvement and efficiency enhancement can be proposed, based on the identified complications.

**Automated approval mechanisms** could be implemented for requests falling below a certain financial threshold or for standard travel types. This approach would effectively bypass manual review by Directors or Budget Owners, the bottlenecks of the system. Given the Pre-Approver's observed efficiency in accelerating processes, **expanding their workload** or assigning them additional responsibilities could also be useful. Furthermore, it could be interesting to understand if certain review or approval steps could be done in **parallel**, rather than sequentially, reducing the overall lead time of the process.
Given the high rework linked to system activities, these specific process loops should be **further investigated** to identify where automation can be improved and errors could be reduced. The goal is to make these already human-independent tasks more efficient, minimizing the time wasted on repeated system cycles.
**Automated validation checks** could be implemented at the point of submission for employees. These checks, focusing for instance on mandatory fields or correct document formats, would be useful for international declarations and permit requests, maybe leading to less rejections and less rework. If the reason for rejections, not known in this study, is related to a lack of knowledge among employees, it could also be useful to provide them with **training** on how to correctly submit different types of declarations.

# Conclusions

The assignments presented in the first part of this report have been successfully completed. The event logs were filtered to reduce noise and irrelevant data and then segmented to differentiate the various roles involved in the processes.

Several key statistics, such as case duration, case size, rework analysis, and rejection rates, along with conformance checking, were calculated across the different logs. These were broken down by segment and aggregated as needed.

This allowed for the comparison of segments across the logs, revealing process complications and identifying potential improvements and optimization opportunities.

The results were obtained using numerous functions from the PM4Py documentation, along with some custom functions specifically developed to achieve particular objectives during the project's development, as presented in previous sections.

In conclusion, this analysis has offered a clear picture of how the organization's operations are currently functioning and what could be improved.

# Code reference

The complete codebase for this report, along with additional material, can be accessed here: https://github.com/haricalzi/Travel-Reimbursement