# XM_0128 - FUNDAMENTALS OF ADAPTIVE SOFTWARE

Name of the Study:

[redacted]

VU
*Vrije* Universiteit
Amsterdam

# EMERGENT WEB SERVER

**TEAM: GREEN LIGHT DISTRICT (2_4)**

| Student Name | Student Number |
|---|---|
| [redacted] | [redacted] |
| [redacted] | [redacted] |
| [redacted] | [redacted] |
| [redacted] | [redacted] |

**GitHub Repositories**

**EWS:** [redacted]
**UPISAS:** [redacted]

## TABLE OF CONTENTS

# SYSTEM DESCRIPTION

The Emergent Web Server (EWS) is a self-adaptive system designed to serve web content dynamically under varying load conditions. It is the managed system within our self-adaptive architecture, with UPISAS acting as the managing system.

EWS is capable of adjusting its server configuration in real-time to handle changing web traffic patterns efficiently. The system achieves this by switching between different architectural configurations, each tailored to specific types of web requests and traffic volumes.

UPISAS manages the adaptive behaviour of EWS by continuously monitoring key performance indicators, primarily the response time of the server. It employs a decision-making process informed by real-time data analysis, leveraging online learning algorithms to strike a balance between exploiting well-performing configurations and exploring potentially better ones.

The interaction between UPISAS and EWS is facilitated through a REST API, which allows for the monitoring, adaptation, and management of the web server. This interface is crucial for integrating advanced machine learning strategies such as the contextual bandit approach, which uses additional contextual information like the number of requests to improve the decision-making process for adaptations.

This self-adaptive system demonstrates an effective approach to maintaining optimal server performance and resource utilization during unpredictable web traffic, ensuring reliability and efficiency in web service delivery.

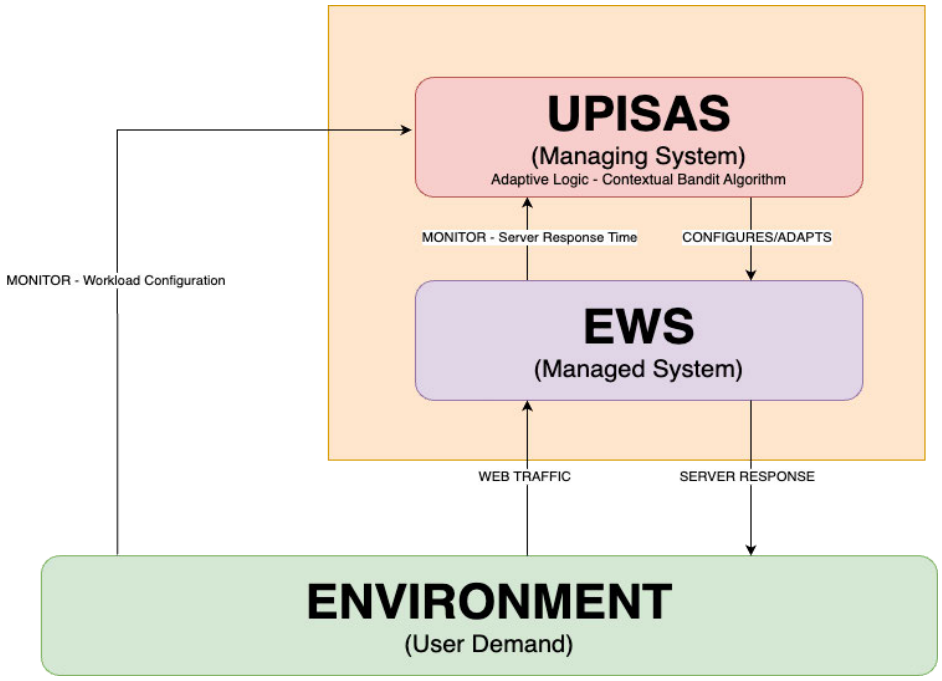


**Fig 1: High Level Diagram of the Emergent Web Server integrated with UPISAS**

# ANALYSIS OF UNCERTAINTIES

2.1

**Name: Response Time Variability**

**Classification:** Execution

**Context:** This uncertainty is a consequence of the variable workload patterns and the server's ability to handle them efficiently.

**Impact:** Fluctuations in response time directly affect user experience and can impact business metrics like conversion rates and bounce rates.

**Degree of Severity:** Moderate to High - Users typically have a low tolerance for slow response times, which can directly affect the perceived quality of the service.

**Sample Illustration:** Under a heavy load of mixed content types, response times can degrade if the server is not quickly adapted to the optimal configuration for the current mix of requests.

**Evaluation:** The effectiveness of the adaptation strategy will be measured by the stability and consistency of response times under varying traffic conditions.

**Also Known As:** Service Quality Variability

2.2

**Name: Workload Pattern Variability**

**Classification:** Execution

**Context:** The uncertainty is rooted in the unpredictable and dynamic nature of user requests, which may vary by type (i.e. text or image content), and frequency.

**Impact:** Inconsistent workload patterns can lead to server configurations that are not aligned with current demand, resulting in inefficiencies and potential performance bottlenecks.

**Degree of Severity:** High - Misalignment between server capacity and demand can cause significant performance issues and user dissatisfaction.

**Sample Illustration:** A sudden spike in image requests can overwhelm a server optimized for text, leading to long response times and a suboptimal user experience.

**Evaluation:** The strategy will be evaluated based on its ability to adapt server configurations dynamically to match real-time workload patterns, aiming to reduce the occurrence of performance bottlenecks.

**Also Known As:** Demand Uncertainty or Traffic Pattern Uncertainty

# REQUIREMENTS

**Traditional Requirements:**

1. **Response Time Management: R1**: The server SHOULD maintain an average response time of less than “X” seconds for all incoming requests, regardless of the workload pattern.
2. **Workload Adaptation: W1**: The server SHOULD adapt its configuration when the incoming request rate exceeds “Y” requests per second.

**RELAX Requirements:**

1. **Response Time Management: R1'**: The server SHALL maintain an average response time AS CLOSE AS POSSIBLE TO less than “X” seconds for all incoming requests, EVEN AS the request rate VARIES.

   **ENVIRONMENT**: current request rate

   **MONITOR**: average response time

   **RELATION**: response time is related to request rate.

2. **Workload Adaptation: W1':** The server SHALL adapt its configuration AS EARLY AS POSSIBLE AFTER the incoming request rate REACHES AS MANY AS “Y” requests per second.

   **ENVIRONMENT**: server load

   **MONITOR**: request rate

   **RELATION**: server configuration is dependent on request rate.

To be more specific, in our implementation, we have taken the value of “X” as 5 milliseconds and “Y” as 100 requests per second.

# ANALYSIS OF POTENTIAL SOLUTIONS:

**Solution: Reinforcement Learning (RL) with Deep Neural Networks**

**Advantages:**

- Can learn complex patterns in high-dimensional spaces.
- Capable of learning optimal strategies through trial and error.
- Adapts to changes in the environment over time.

**Limitations:**

- Requires significant computational resources for training.
- Needs a large amount of diverse training data to perform well.
- May be overkill for simple adaptation problems and can be difficult to interpret.

**Decision: Rejected**

**Reason:**
The computational overhead and complexity of deep RL may not justify the incremental gains for this application. Simpler models may suffice and be more interpretable and maintainable.

**Solution: Genetic Algorithms (GA) for Configuration Optimization**

**Advantages:**
- Can optimize over a large space of possible configurations.
- Does not require gradient information or differentiability of the model.
- Can escape local optima through evolutionary strategies.

**Limitations:**
- Can be slow to converge to an optimal solution.
- Requires careful tuning of evolutionary parameters (mutation rate, selection pressure, etc.).
- Performance can be highly variable and unpredictable.

**Decision:** **Considered**

**Reason:**
GAs offer a robust search mechanism for configuration spaces but may require extensive computational time. They are a potential solution if the search space is complex and other methods fail.

**Solution: Transfer Learning with Pre-trained Models**

**Advantages:**
- Leverages existing models trained on large datasets to improve performance.
- Reduces the need for extensive domain-specific data.
- Can quickly adapt to the specific task with fine-tuning.

**Limitations:**
- The effectiveness depends on the similarity between the source and target tasks.
- Pre-trained models can be large and require significant resources to fine-tune and deploy.
- There may be legal and ethical considerations around using pre-trained models.

**Decision:** **Considered**

**Reason:**
Transfer learning is a viable option if there are pre-existing models closely related to the task at hand. It can jump-start the learning process but requires careful consideration of model applicability.

**Solution: Semi-Supervised Learning**

**Advantages:**
- Utilizes both labeled and unlabeled data, making it useful when labels are scarce or expensive to obtain.

- Can improve learning accuracy when large amounts of unlabeled data are available.
- Bridges the gap between supervised and unsupervised learning.

**Limitations:**

- Model performance can suffer if the assumptions about the unlabeled data do not hold.
- The approach can be sensitive to the quality of the labeled data.
- There is an additional complexity in designing semi-supervised learning algorithms.

**Decision:** **Considered**

**Reason:**
If obtaining labeled data is challenging, semi-supervised learning can be an efficient way to leverage the available data. However, the assumptions about the data need to be verified.

**Solution Name: Contextual Bandit-Based Adaptive Configuration Strategy**

**Advantages:**

- Integrates the robustness of contextual bandits, which allows for better decision-making by considering additional context such as workload configurations and user demand, enhancing the relevance of each configuration choice.
- Employs the AdaptiveGreedy algorithm from reinforcement learning, suitable for continuously fine-tuning the system to adapt to the changing web traffic and server demands.
- The strategy's online learning mechanism facilitates real-time updates to the model, essential for the dynamic and unpredictable nature of web server workloads.

**Limitations**:

- Requires a strategic approach to feature engineering to ensure that the model captures the full breadth of the operational context.
- The necessity for proper tuning of the exploration-exploitation trade-off poses a challenge, particularly in maintaining system performance while exploring new configurations.
- The efficacy of the adaptation strategy depends on the availability of precise and timely feedback on the impact of configuration changes.

**Decision:** **Selected**

**Reason:**
The selection of the Contextual Bandit-Based Adaptive Configuration Strategy is due to its ability to leverage additional contextual data, a critical factor in the complex environment of web server management. The strategy's reinforcement learning core, particularly the implementation of the AdaptiveGreedy algorithm, is tailored to address the needs of a system where operational conditions are not static. Despite the challenges associated with feature engineering and parameter tuning, this approach is poised to significantly enhance EWS's responsiveness and efficiency. The contextual bandit model's inherent capability to balance the need for exploration with the effectiveness of exploitation strategies makes it an ideal choice for managing the EWS's dynamic conditions, promoting ongoing optimization and learning.

# PROPOSED ADAPTATION STRATEGY:

The proposed adaptation strategy for the Emergent Web Server (EWS) involves a contextual multi-armed bandit algorithm specifically the AdaptiveGreedy algorithm within the UPISAS framework. This strategy aims to optimize the server's response time and resource utilization by dynamically adjusting its configuration based on current web traffic patterns and request types.

**Adaptation Logic (MAPE-K Framework):**

**Monitoring:**

- UPISAS is tasked with the continuous monitoring of EWS. It collects metrics such as workload configuration (text or image), volume of traffic, and server response times.
- The data gathered is structured into a contextual format that can be effectively analyzed by the contextual bandit algorithm to determine the current state of the system and its environment.

**Analysis:**

- At this stage, the Adaptive Greedy algorithm assesses the monitored data to gauge the effectiveness of various server configurations within the current context.
- It is here that a decision is made on whether a change in server configuration could potentially improve performance, based on an estimation of expected rewards versus the current configuration.
- The output of this phase is a decision on whether to maintain the current configuration or to proceed to the Planning phase for adaptation.

**Planning:**

- Triggered by the Analysis phase's directive, the Planning phase involves generating a set of actionable server configuration changes that could enhance the system’s performance.
- These actions are devised with the objective of optimizing expected rewards, which is a combination of maintaining high-performing configurations (exploitation) and trying out new configurations (exploration).
- The planning phase is pivotal to the adaptation strategy as it maps out a course of action that is anticipated to yield the highest reward in the operational context of the EWS.

**Execution:**

- The selected server configuration from the Planning phase is then executed, applying the changes to EWS.
- UPISAS ensures that the transition to the new configuration aims for minimal service interruption and maintaining system stability.

**Knowledge:**

- The Knowledge component is where the results of the executed actions are fed back into the system, updating UPISAS's understanding of configuration performance.
- This feedback loop is essential for refining the predictive model within the contextual bandit algorithm, leading to improved decision-making in future iterations.

The synthesis of these phases forms a cohesive adaptive strategy that enables EWS to maintain an optimal operational stance in the face of diverse and dynamic web service demands. The strategy is supported by the principle of continuous improvement, ensuring that the EWS's service delivery is both responsive and efficient, adapting in real-time to the changing patterns of user interaction and system stress. The activities in the adaptation strategy are illustrated in Fig 2 below.

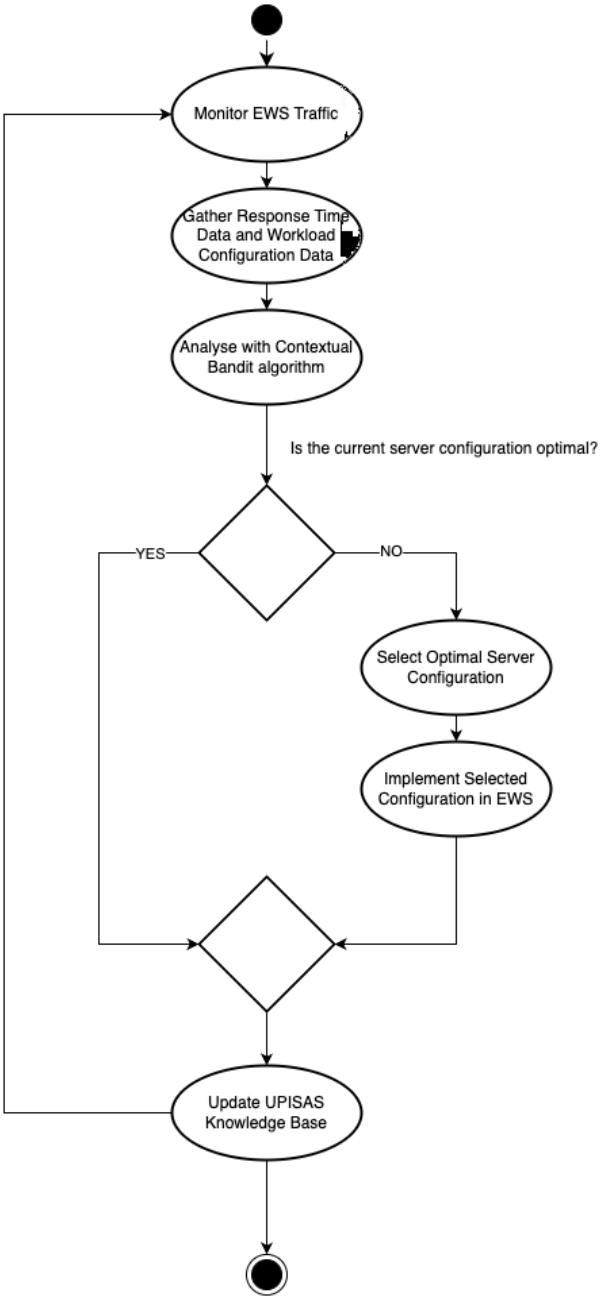


**Fig 2: Activity Diagram of our adaptation strategy showcasing the MAPE-K loop**

**PSEUDOCODE:**

```
initialize UPISAS
while true:
   // Monitoring
   traffic_data = monitor_EWS_traffic()
   request_types = classify_requests(traffic_data)
   response_times = calculate_response_times(traffic_data)
   workload_data = gather_workload_data(traffic_data)
   current_context = create_context(request_types, response_times, workload_data)
   // Analysis
   if is_adaptation_needed(current_context):
      // Planning
      optimal_config, expected_reward = plan_optimal_configuration(current_context)
      // Execution
      if expected_reward indicates significant improvement:
         execute_configuration_change(EWS, optimal_config)
      else:
         maintain_current_configuration()
   else:
      maintain_current_configuration()
   // Knowledge Update
   update_knowledge_base(traffic_data, response_times, workload_data, optimal_config)
   // Wait for the next monitoring cycle
   wait_for_next_cycle()

// Support functions
function monitor_EWS_traffic():
   // Implementation to collect traffic and workload data from EWS

function classify_requests(traffic_data):
   // Implementation to classify requests into types (text/image)
function calculate_response_times(traffic_data):
   // Implementation to calculate response times from EWS data

function gather_workload_data(traffic_data):
   // Implementation to gather additional workload data relevant for the context
function create_context(request_types, response_times, workload_data):
   // Implementation to combine request types, response times, and workload data into a context

function is_adaptation_needed(context):
   // Use contextual bandit algorithm to analyze current context
   // Returns True if a new configuration could potentially improve performance

function plan_optimal_configuration(context):
 // Use contextual bandit algorithm to determine the optimal server configuration and expected reward
function execute_configuration_change(EWS, new_config):
   // Implementation to change the server configuration of EWS

function maintain_current_configuration():
   // Keep the current configuration unchanged
function update_knowledge_base(traffic_data, response_times, workload_data, new_config):
   // Update the system's knowledge base with the effects of the new configuration

function wait_for_next_cycle():
   // Implementation to wait before starting the next monitoring cycle
```

**Expected Effects of the Adaptation Strategy:**

**Expected Benefits:**

- **Optimized Server Performance:** The strategy will enable EWS to adapt its configuration dynamically, leading to improved response times and overall server performance.
- **Efficient Resource Utilization:** By aligning server configuration with the current demand, the system can use resources more efficiently, potentially lowering operational costs.
- **Enhanced Responsiveness to Traffic Variations:** The server's ability to respond quickly to changes in web traffic patterns will be significantly improved.
- **Learning and Improvement Over Time:** As the system continues to gather data, its decision-making process will become more refined and accurate.

**Expected Side-Effects:**

- **Initial Learning Phase:** There may be a period of suboptimal performance initially as the system gathers sufficient data to make informed decisions.
- **Exploration Risks:** The exploration aspect of the bandit algorithm might lead to temporary performance dips when testing less-optimal configurations.
- **Complexity in Implementation:** Integrating a sophisticated algorithm like a contextual bandit into an existing system like EWS presents inherent complexities and potential integration challenges.
- **Adjustment to Dynamic Learning:** The system's continuous learning approach may necessitate periodic adjustments and fine-tuning to maintain optimal performance.

# IMPLEMENTATION:

**Contextual Adaptive Greedy Approach for UPISAS-EWS Integration**

**Overview:**

The adaptation strategy implemented for UPISAS within the context of the Emergent Web Server (EWS) leverages a sophisticated contextual bandit approach, specifically the Adaptive Greedy algorithm, to dynamically optimize server performance. This method is selected for its ability to adaptively respond to real-time data while balancing exploration and exploitation.

**Design:**

**Feature Engineering and Preparation:**

The adaptation begins with a robust feature engineering process. This includes creating ratios and differences between types of clients and integrating contextual features like request type and server configuration.

```
# Feature engineering
data['Text_to_Total_Ratio'] = data['Text Clients'] / data['Total Clients']
data['Image_to_Total_Ratio'] = data['Image Clients'] / data['Total Clients']
data['Text_vs_Image'] = data['Text Clients'] - data['Image Clients']
data['Avg_Response_Time'] = data['Response Time'] / data['Counter']

# Adding context features
context_features = ['Request', 'Http', 'Compression', 'Cache']
feature_columns = ['Total Clients', 'Text Clients', 'Image Clients',
                   'Text_to_Total_Ratio', 'Image_to_Total_Ratio', 'Text_vs_Image'] + context_features
features = data[feature_columns]
features = features.to_numpy()
```

This step is crucial for capturing the interplay of web traffic and server load in a format that the adaptive model can utilize effectively.

**Model Training with Adaptive Greedy:**

An Adaptive Greedy model from the *contextualbandits* library is trained on the engineered features. This model is particularly adept at handling the multi-armed bandit problem in a context-aware manner.

```
# Using Contextual Bandith Alogorithms
self.model = AdaptiveGreedy(deepcopy(base_ols), nchoices = choices_array,
                            smoothing = (1,2), beta_prior = None,
                            decay_type = 'percentile', decay = 0.9997, batch_train = True,
                            random_state = 4444)
```

The model is continuously updated with batch training, ensuring that it learns and adapts from new data patterns in real-time.

**Analyzing and Planning for Configuration Changes:**

The analyze and plan methods in the strategy play a pivotal role in decision-making. They evaluate the current performance and determine the necessity and nature of configuration changes.
*(Note: The analyze phase uses a threshold to decide on configuration changes. This part may need adjustments on MacOS, as EWS tends to run slower on MacOS. It might be necessary to lower the threshold values for proper performance on these systems.)*

```
def analyze(self):
    print("------------------ ANALYZE ------------------")
    data = self.knowledge.monitored_data
    data = pd.DataFrame([self.knowledge.monitored_data])
    counter = int(data.at[0, "Counter"])
    avg_response_time = data.at[0, "Response Time"] / data.at[0, "Counter"]

    print(f"[Analysis]\tAverage Response Time: {avg_response_time}\tCounter: {counter}")

    # Check against the thresholds to decide if the system needs adaptation
    # The counter threshold is set to 1000 because monitor data is collected over a 10 second interval.
    # Essentially the threshold is 100 per second.
    if (avg_response_time > 5 or counter > 1000):
        self.knowledge.analysis_data["avg_response_time"] = avg_response_time
        self.knowledge.analysis_data["counter"] = counter
        return True
    return False
```

Based on this analysis, the system plans the optimal server configuration, aiming to enhance performance metrics such as response times.

```
    # Accessing the monitored data to populate X_test
    X_test.at[0,'Total Clients'] = data.at[0,'Total Clients']
    X_test.at[0,'Text Clients'] = data.at[0,'Text Clients']
    X_test.at[0,'Image Clients'] = data.at[0,'Image Clients']
    X_test.at[0,'Text_to_Total_Ratio'] = data.at[0,'Text_to_Total_Ratio']
    X_test.at[0,'Image_to_Total_Ratio'] = data.at[0,'Image_to_Total_Ratio']
    X_test.at[0,'Text_vs_Image'] = data.at[0,'Text_vs_Image']
    X_test.at[0,'Request'] = configs.loc[self.current_config, 'Request']
    X_test.at[0,'Http'] = configs.loc[self.current_config, 'Http']
    X_test.at[0,'Compression'] = configs.loc[self.current_config, 'Compression']
    X_test.at[0,'Cache'] = configs.loc[self.current_config, 'Cache']

    self.model.partial_fit(X_test, np.array([self.current_config]),
                           np.array([self.exponential_decay_reward(data.at[0,'Avg_Response_Time'])]))

    config_id = int((self.model.predict(X_test))[0])
```

**Execution and Continuous Learning:**

The planned configuration is executed, with the system applying changes to the EWS. This step is critical for realizing the benefits of the adaptive strategy.

The system's knowledge base is continuously updated with the outcomes of these actions, allowing for an evolving understanding of performance under various configurations.

```
config_composition_row = configs[configs['ID'] == config_id]
config_composition =  config_composition_row['Composition'].values[0]

self.knowledge.plan_data = { "id" : config_id,
                "config": config_composition }
```

**Technologies and Libraries Used**

**Python:** The core programming language for the project. It is versatile and has strong support in data science and machine learning communities.

**Pandas**: Utilized for data manipulation and analysis, this library is instrumental in handling and preparing datasets for the learning algorithms.

**NumPy**: Predominantly used for scientific computing. It offers comprehensive support for array and matrix operations, which are essential in handling numerical data for machine learning tasks.

**Scikit-learn:** A key open-source machine learning library in Python. It provides a range of tools for machine learning and statistical modeling including classification, regression, clustering, and dimensionality reduction, and is known for its ease of use and flexibility.

**ContextualBandits:** This library is crucial for implementing the online learning strategies in the project. It offers a range of algorithms for contextual multi-armed bandit problems, including the Adaptive Greedy algorithm used in the project. It complements the machine learning capabilities of Scikit-learn, specifically catering to the needs of online, data-driven decision-making.

**Deepcopy (from the Python Standard Library):** Used to ensure that copies of objects are recursively made, preserving the nested structures within the project's data, crucial for maintaining the integrity of models during training.

**Implementation Process: Prototype-Based Development - UPISAS Extension**

**Conceptualization and Goal Setting:**

- The project started with a clear objective: to extend UPISAS as an effective managing system for EWS.
- Initial brainstorming sessions focused on identifying key functionalities and interdependencies between UPISAS and EWS.

**Prototype Creation:**

- A basic prototype was developed, containing the basic structure of UPISAS as a managing system for EWS.
- This early version aimed to establish a communication framework between UPISAS and EWS and to integrate a basic version of the multi-arm bandit algorithm.

**Iterative Development and Refinement:**

- The development process was iterative, with each cycle enhancing the prototype's capabilities and refining its integration with EWS.
- Context was introduced and the algorithm was tuned to work well with the different contexts on a trial-and-error basis.
- Special attention was given to the algorithm's effectiveness in managing EWS configurations based on web traffic data.

**Feedback Incorporation and Adaptation:**

- Feedback, primarily from internal testing, was crucial in shaping the prototype's evolution.
- This feedback led to adjustments in the managing strategies and improved the overall efficacy of UPISAS in managing EWS.

**Rigorous Testing and Validation:**

- The prototype was subjected to extensive testing, ensuring that it met the requirements of effectively managing EWS.
- Tests focused on the prototype's ability to dynamically optimize EWS's performance and resource utilization.

**Evolution of the Prototype:**

- Throughout the development, the prototype evolved significantly, with the team incorporating advanced features such as real-time data processing and sophisticated decision-making algorithms.
- This evolution represented the growing sophistication of UPISAS as a managing system.

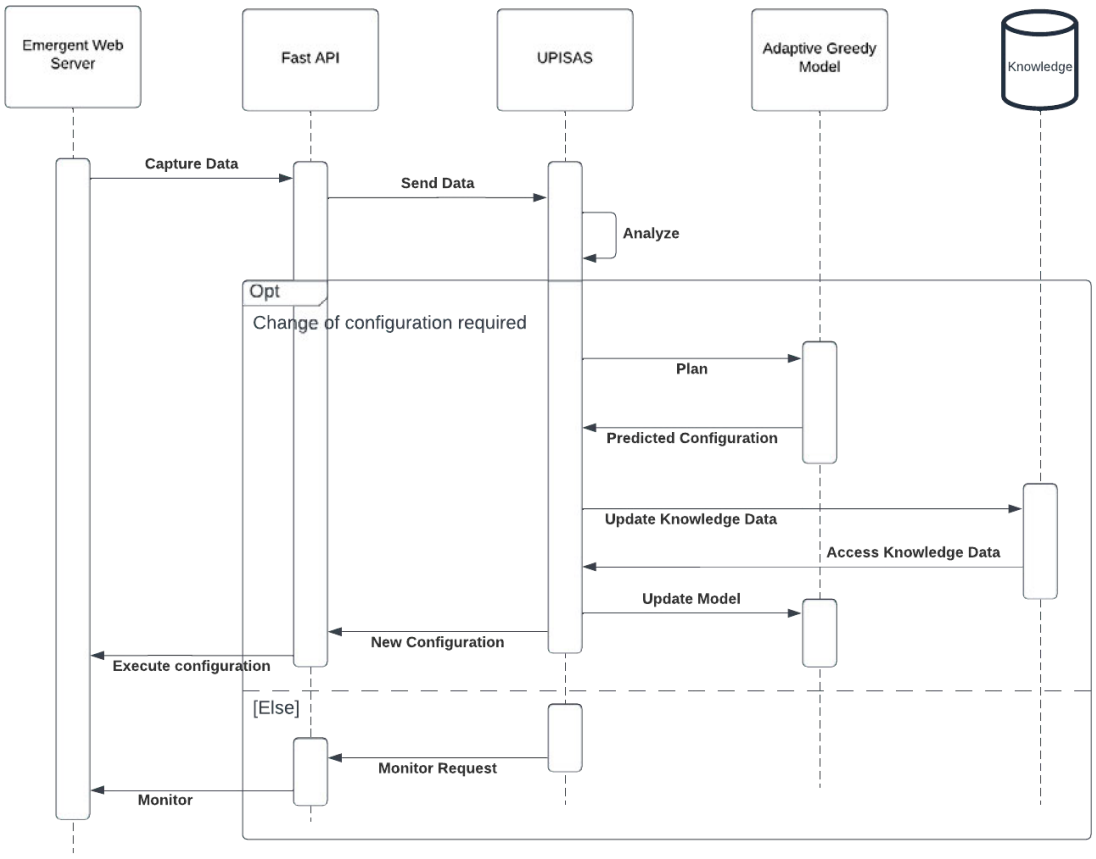


**Fig 3: Sequence Diagram of our adaptation strategy**

The sequence diagram in Fig 3 illustrates the operational workflow within the adaptive system. It starts with data capture by the Emergent Web Server, followed by data analysis via UPISAS to determine if a configuration change is needed. If affirmative, the Adaptive Greedy Model devises a new configuration, which is then executed, with the outcomes recorded in the knowledge base. This continuous monitoring and updating loop ensures the system's adaptability and performance optimization.

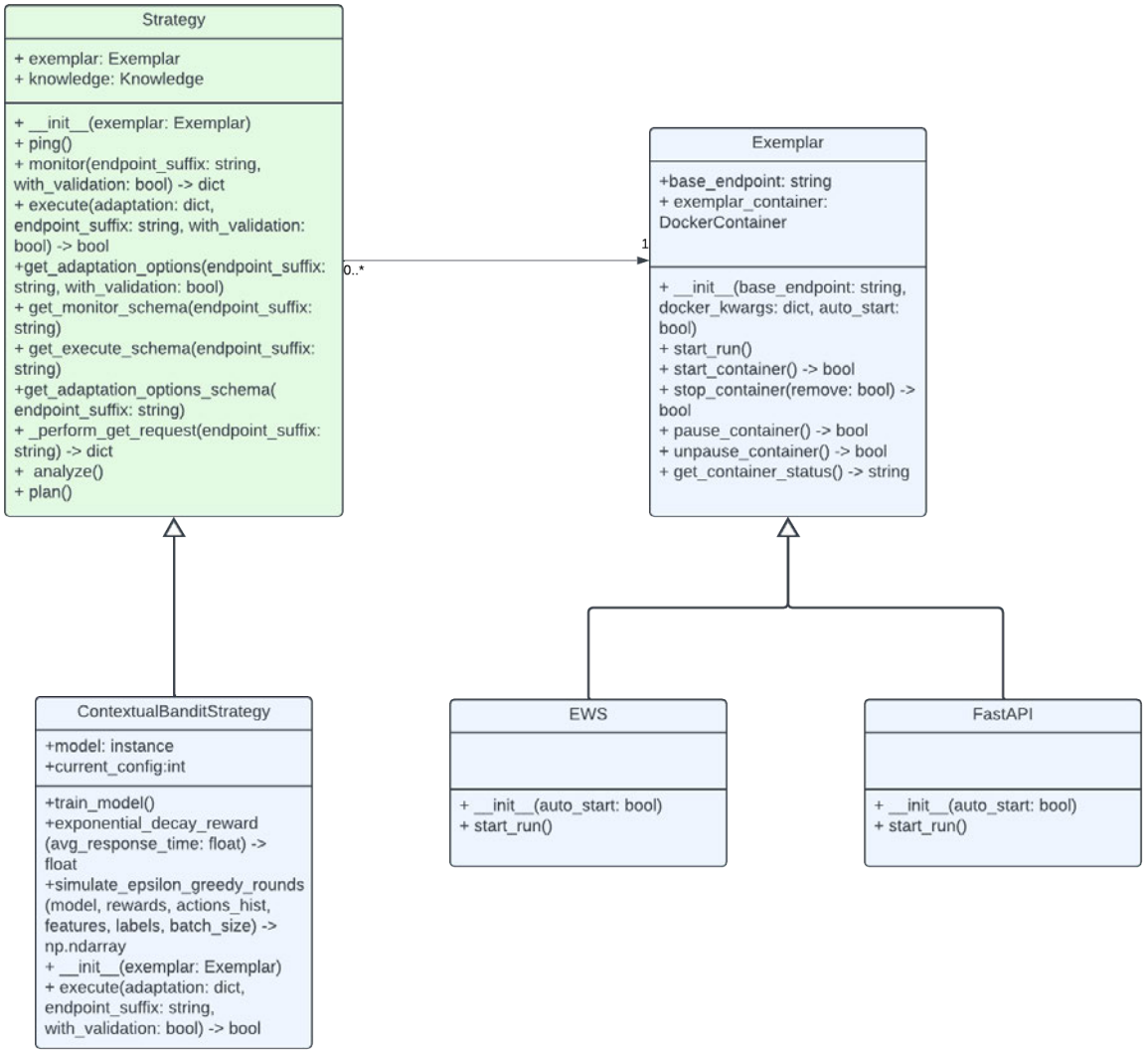


**Fig 4: Class Diagram of our implementation of the Contextual Bandit adaptation strategy**

The class diagram in Fig 4 illustrates the object-oriented structure of the system. At the top level, 'Exemplar' is an abstract base class that provides an interface for container management and defines common attributes. Derived from this are 'EWS' and 'FastAPI', which implement the specifics of starting and managing web server instances. The 'Strategy' class serves as an abstract base for adaptation strategies, defining common operations like monitoring, execution, and planning. 'ContextualBanditStrategy' is a concrete implementation that specializes the 'Strategy' class, incorporating machine learning models and methods for reward calculation and strategy execution. This setup encapsulates the functionality and promotes reusability and maintainability in the system's design.

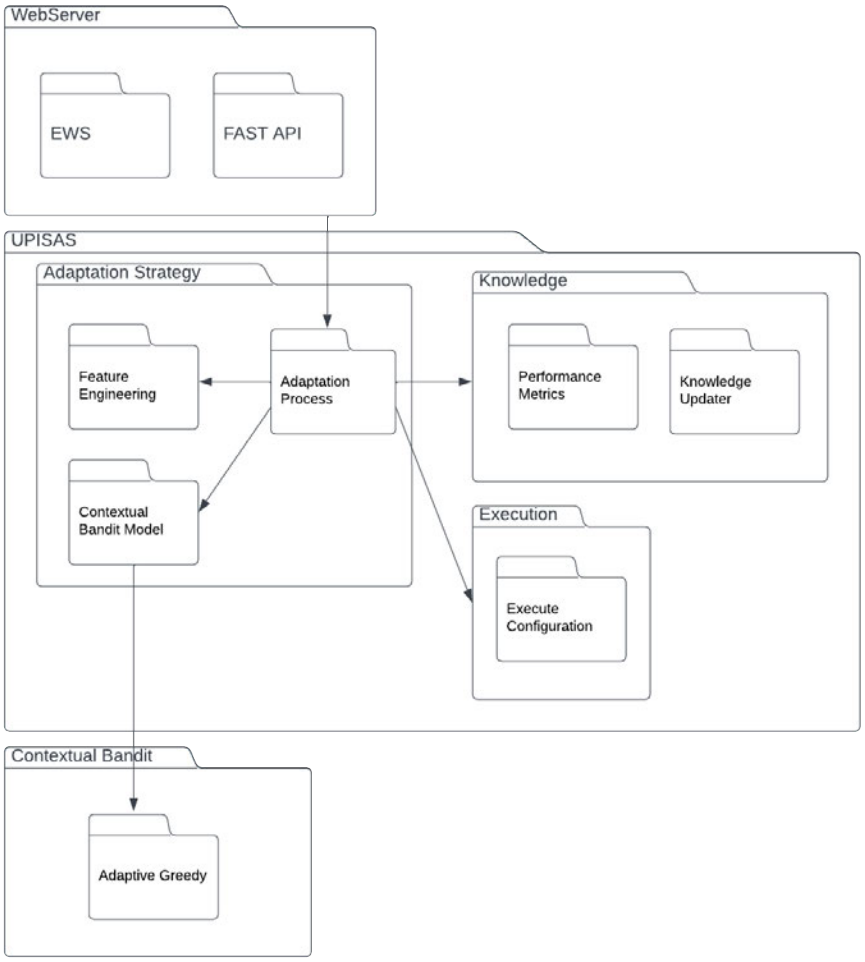


**Fig 5: Package Diagram of our implementation of the Contextual Bandit adaptation strategy**

The package diagram in Fig 5 presents the modular structure of the system. The 'WebServer' package contains two sub-modules: 'EWS' and 'FAST API', which handle web service operations. 'UPISAS' serves as the decision-making core, with 'Feature Engineering' refining data inputs and 'Adaptation Process' determining configuration changes, guided by the 'Contextual Bandit Model'—specifically, the 'Adaptive Greedy' algorithm. The 'Knowledge' package stores performance metrics and updates the model, while 'Execution' is responsible for implementing the chosen configurations. This architecture ensures a cohesive, well-orchestrated adaptation strategy.

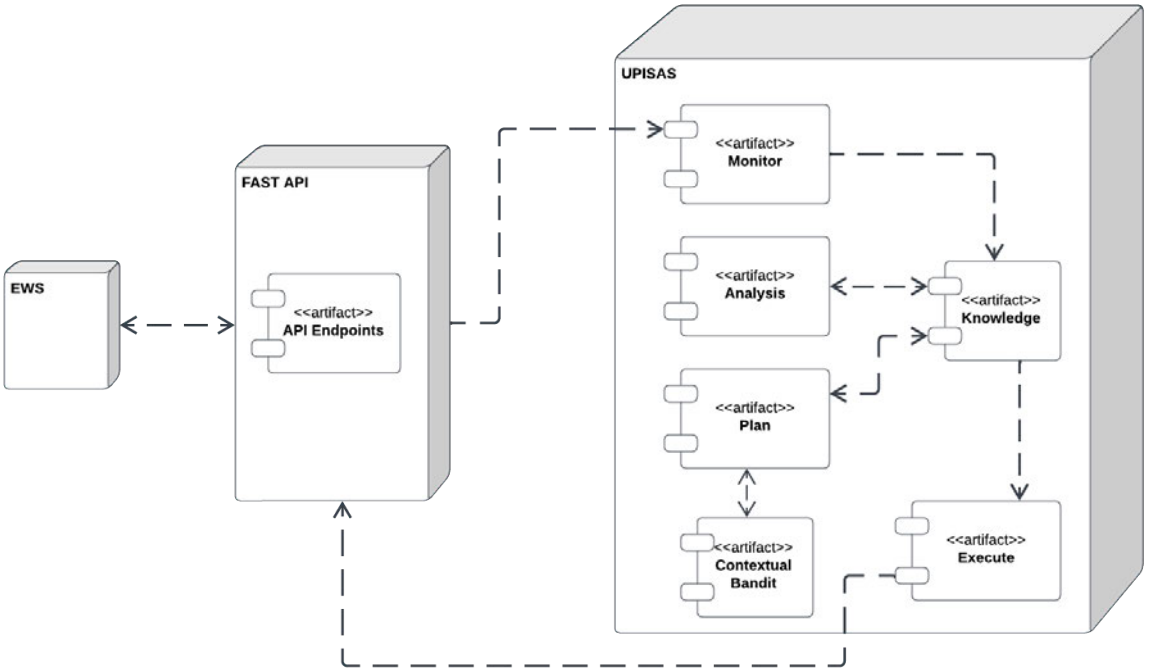


**Fig 6: Deployment Diagram of our implementation of the Contextual Bandit adaptation strategy**

The deployment diagram in Fig 6 visualizes the UPISAS system's architecture, highlighting the data flow and interactions between the Emergent Web Server (EWS), FAST API, and UPISAS. The EWS node captures server data, which is communicated through FAST API's endpoints. UPISAS is depicted as a central node where monitoring, analysis, planning, and execution processes take place, supported by a knowledge base for informed decision-making. The Contextual Bandit module within UPISAS, particularly the Adaptive Greedy component, is central to the adaptive strategy, determining optimal server configurations in response to real-time data.

# ALTERNATE IMPLEMENTATIONS:

Some of the alternate implementations considered for our adaptation strategy are discussed below

**Alternate Implementation-1: Stochastic Gradient Descent (SGD) with Epsilon Greedy Algorithm**

1. **Data Preparation and Feature Engineering:**

- The initial steps involved importing and preprocessing the data from train_data.csv.
- Feature engineering was applied to create more insightful features, such as the ratio of text to total clients and image to total clients, providing a more nuanced understanding of the traffic composition.

```
# Import and prepare the data
data_train = pd.read_csv('../../train_data.csv')
data_train.fillna(0, inplace=True)  # Handling missing values

# Feature engineering to create more informative features
# This improves the model's ability to learn from the data
data_train['Text_to_Total_Ratio'] = data_train['Text Clients'] / data_train['Total Clients']
data_train['Image_to_Total_Ratio'] = data_train['Image Clients'] / data_train['Total Clients']
data_train['Text_vs_Image'] = data_train['Text Clients'] - data_train['Image Clients']
data_train['Avg_Response_Time'] = data_train['Response Time'] / data_train['Counter']
```

2. **Model Training with SGD Classifier:**

- A classifier was trained using the SGD method to predict optimal server configurations.
- The classifier was incrementally trained in batches for efficiency, considering the dynamic nature of the web traffic.

```
# Initialize the SGD Classifier
model = SGDClassifier(max_iter=1000, tol=1e-3)

# Training the model in batches for efficiency
batch_size = 50
for i in range(0, len(X_train), batch_size):
    batch_end = min(i + batch_size, len(X_train))
    model.partial_fit(X_train[i:batch_end], ids_train[i:batch_end], classes=np.unique(config_ids))
```

3. **Epsilon-Greedy Strategy:**

- An epsilon-greedy strategy was implemented to balance the exploration of new configurations with the exploitation of known efficient ones.
- This decision-making process was crucial for adapting to varying traffic conditions while avoiding stagnation.

```
for i in range(len(X_test)):
    # Random exploration vs. exploitation based on the epsilon value
    if np.random.rand() < epsilon:
        # Exploration: Randomly choose a Config ID
        pred_config_id = np.random.choice(data_train['Config ID'])
    else:
        # Exploitation: Choose the best Config ID based on the model's prediction
        pred_config_id = model.predict([X_test[i]])[0]
```

4. **Reward Calculation and Model Update:**

- A reward function based on average response time was defined to evaluate the effectiveness of chosen configurations.
- The model was updated using a feedback loop where the rewards were translated into sample weights, enhancing the learning process.

```python
def exponential_decay_reward(self, avg_response_time):
    scale_factor = 0.0001
    return np.exp(-avg_response_time * scale_factor) if avg_response_time > 0 else 0

def rewards_to_weights(self, rewards):
    rewards = np.array(rewards)
    rewards = (rewards - rewards.min()) / (rewards.max() - rewards.min())
    rewards += 0.1  # Avoid zero weights
    return rewards
```

```python
def predict_and_update(self):
    predicted_rewards = []
    predicted_config_ids = []

    for i in range(len(self.X_test)):
        if np.random.rand() < self.epsilon:
            # Exploration
            pred_config_id = np.random.choice(self.ids_train.unique())
        else:
            # Exploitation
            pred_config_id = self.model.predict([self.X_test[i]])[0]

        reward = self.exponential_decay_reward(self.avg_response_time_per_config_id[pred_config_id])
        predicted_rewards.append(reward)
        predicted_config_ids.append(pred_config_id)

    # Convert rewards to sample weights for training
    reward_weights = self.rewards_to_weights(predicted_rewards)

    # Update the model
    self.model.partial_fit(self.X_test, self.ids_test, sample_weight=reward_weights)

    return predicted_config_ids
```

This approach was considered to leverage the strengths of SGD classifiers in handling large and dynamic datasets typical in web traffic analysis. The epsilon-greedy strategy added an element of real-time adaptability, allowing the system to explore and exploit configurations effectively.

**Limitations:**

The primary limitation of this approach was the potential for slower convergence and suboptimal performance during the initial learning phase. Furthermore, the reliance on a classifier model might not capture the complex, non-linear relationships within the web traffic data as effectively as other machine learning techniques like Random Forest

**Alternate Implementation-2: RandomForestRegressor with Enhanced Feature Engineering**

1. **Data Preparation and Feature Engineering:**

- The strategy involved preprocessing data from train_data_with_comp.csv, focusing on creating a comprehensive set of features to understand the web traffic thoroughly.

- Advanced feature engineering was applied to capture the nuances of web traffic. This included ratios, differences, and interaction terms to provide a detailed view of client patterns and their impact on server response times.

```
data['Text_to_Total_Ratio'] = data['Text Clients'] / data['Total Clients']
data['Image_to_Total_Ratio'] = data['Image Clients'] / data['Total Clients']
data['Text_vs_Image'] = data['Text Clients'] - data['Image Clients']
data['Avg_Response_Time'] = data['Response Time'] / data['Counter']

context_features = ['Request', 'Http', 'Compression', 'Cache']
feature_columns = ['Total Clients', 'Text Clients', 'Image Clients',
                   'Text_to_Total_Ratio', 'Image_to_Total_Ratio',
                   'Text_vs_Image', 'Config ID',] + context_features
features = data[feature_columns]
avg_y = data['Avg_Response_Time']
```

2. **Random Forest Model Training:**

- A RandomForestRegressor was employed to predict optimal server configurations, leveraging the extensively engineered features to model complex relationships in the data.
- This model choice aimed to leverage its ability to handle intricate patterns in web traffic data for predicting server response times.

```
self.model = RandomForestRegressor(n_estimators=100, random_state=42)
self.model.fit(features_scaled, avg_y)
```

3. **Analysis Phase:**

- The analysis phase involved checking if the current average response time exceeded a certain threshold, indicating the need for adaptation.

```
# Check against the thresholds to decide if the system needs adaptation
if (avg_response_time > 5 or counter > 1000):
    self.knowledge.analysis_data["avg_response_time"] = avg_response_time
    self.knowledge.analysis_data["counter"] = counter
    return True
return False
```

4. **Planning with Reward Calculation:**

- The planning phase utilized the RandomForest model's predictions to select the configuration with the highest predicted reward.
- Rewards were calculated based on average response times, aligning with the goal of optimizing server performance.

```
def calculate_reward(self, avg_response_time):
    # Define a scale factor for the reward calculation
    scale_factor = 0.0001
    # Calculate the reward based on average response time
    reward = np.exp(-avg_response_time * scale_factor)
    return reward
```

```
rewards = np.array([self.calculate_reward(time) for time in avg_response_time])

# Find the index of the configuration with the highest reward
max_reward_index = np.argmax(rewards)
max_reward_value = rewards[max_reward_index]
# Select the configuration with the highest reward
selected_config_id = configs.loc[max_reward_index, 'ID']
```

This alternative was considered due to the RandomForestRegressor's effectiveness in handling complex datasets and providing accurate predictions. The implementation was aimed at enhancing server adaptability by leveraging a robust machine learning model that could discern subtle patterns in web traffic data.

**Limitations:**

The primary limitation of this approach was the lack of online learning capabilities, as RandomForestRegressor does not support incremental learning or partial_fit. This aspect made it less suitable for environments where data continuously evolves, such as in web traffic.

# SHOWCASE

Please click the link below to view a video of the detailed showcase of our implementation:

EWS Detailed Adaptation Strategy Showcase

# EVALUATION

Our adaptation strategy employs the Adaptive Greedy algorithm, which has been compared against the LinUCB algorithm as our baseline and the Epsilon Greedy strategy for a comprehensive evaluation. The performance metrics are based on the average response time across different client request scenarios.

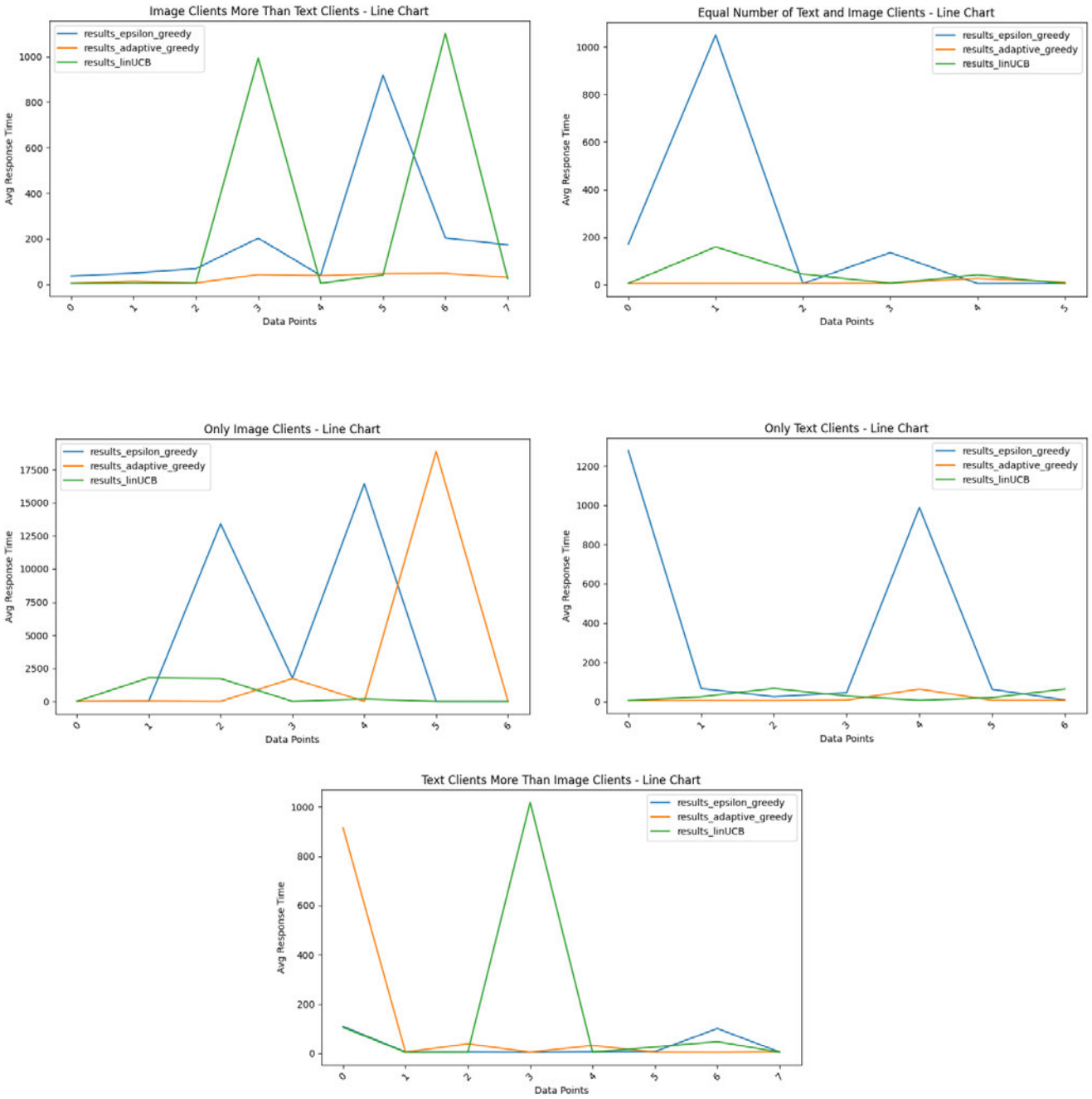


**Fig 7: The above figure shows a line graph for comparison of the adaptive greedy algorithm vs epsilon greedy algorithm vs LinUCB (baseline) while evaluating our adaptation strategy.**

**Quantitative Analysis**

**Equal Number of Text and Image Clients:**
The Adaptive Greedy strategy outperforms both Epsilon Greedy and LinUCB, maintaining a consistently lower average response time consistently. This demonstrates its robustness and effectiveness in a balanced workload environment.

**Image Clients More Than Text Clients:**
In scenarios with a predominance of image requests, the Adaptive Greedy strategy again shows superior performance, minimizing response times effectively. This indicates its capability to adapt to heavy workloads and resource-intensive tasks.

**Only Image Clients:**
The Adaptive Greedy strategy initially had consistently lower response time compared to the baseline, indicating its efficiency in handling homogeneous, high-volume traffic excluding the spike towards the end.

**Only Text Clients:**
With a workload consisting solely of text-based requests, the Adaptive Greedy strategy again outperforms the baseline, though all strategies manage to keep response times relatively low, given the less resource-intensive nature of text processing.

**Text Clients More Than Image Clients:**
In scenarios where text requests are more frequent than image requests, the Adaptive Greedy strategy maintains lower response times despite the initial spike, suggesting its agility in dealing with fluctuating workloads and varying request types.

**Critical Evaluation**

**Advantages:**

- The Adaptive Greedy strategy excels in environments with diverse and dynamic request patterns, adjusting configurations in real-time to optimize performance.
- It demonstrates a notable reduction in response times across all scenarios, despite a few anomalous data points.
- Unlike the LinUCB and Epsilon Greedy strategies, Adaptive Greedy provides a balance between exploration and exploitation without the need for parameter tuning, making it more practical for real-world applications.

**Limitations:**

- The strategy may require a warm-up period to gather sufficient data for accurate decision-making, which could be a constraint in systems with sporadic or non-continuous traffic.
- While the Adaptive Greedy strategy generally performs well, it may still be susceptible to fluctuations in performance during extreme traffic conditions or unprecedented request patterns.

**Conclusion**

The empirical evidence from our evaluations strongly supports the adoption of the Adaptive Greedy strategy in our web server system. It has demonstrated not only the ability to reduce average response times but also adaptability to various client request scenarios, confirming its suitability for dynamic web server environments.

# REFLECTIONS:

Our journey in extending the Emergent Web Server (EWS) with Contextual Bandits and Online Learning within the UPISAS framework has been a blend of discovery and challenge. This project not only deepened our understanding of advanced machine-learning techniques but also tested our abilities to integrate these into a self-adaptive system.

**Learnings:**

**Exploring Contextual Bandits and Feature Engineering:**
Our project initiated an investigation into contextual bandits, highlighting their relevance in decision-making based on web traffic and request types. The process of feature engineering was instrumental in translating raw data into a format usable for our models, providing insights into how server configurations affect response times.

**Online Learning Dynamics and Model Experimentation:**
Our engagement with online learning algorithms underscored their adaptability to real-time data. We experimented with various models, including regression, random forests, and classifiers, to evaluate their effectiveness in the context of dynamic web traffic. This phase involved a critical analysis of each model's strengths and weaknesses concerning online learning applications.

**Integration Challenges and Achievements:**
The integration of sophisticated machine learning algorithms into UPISAS presented multiple challenges. Addressing system compatibility and ensuring effective communication between UPISAS and EWS were pivotal aspects of this process. The successful integration of these elements was a key accomplishment, demonstrating our capability to merge the learning algorithms with the existing system.

**Balancing Exploration and Exploitation:**
A significant aspect of our project was managing the balance between exploring new configurations and exploiting known efficient ones. This balance was essential for ongoing system improvement, ensuring the stability and performance of the system.

**Challenges:**

**Algorithm Selection and Adaptation:**
Selecting an appropriate contextual bandit algorithm was a complex task, driven by the unique requirements of EWS. We evaluated several algorithms, focusing on their adaptability to changing web traffic conditions. This process required a good understanding of each algorithm's functionality and its applicability to our context.

**Complexities in Data Preprocessing:**
Data preprocessing presented its own set of challenges, requiring precision in formatting and feature integration. This task emphasized the intricacies of working with real-world data, highlighting the importance of accuracy in our preprocessing methodology.

**Fine-Tuning Algorithm Parameters:**
Optimizing algorithm parameters, particularly the exploration rate, was a critical task. This optimization was essential for ensuring a balance between effective learning from new data and making efficient decisions based on existing knowledge.

**System Integration and Stability:**
Integrating the chosen algorithm within the existing infrastructure of UPISAS and EWS required a high level of technical proficiency. Our focus was to maintain system stability and ensure seamless internal communication, which were crucial for the successful implementation of our strategy.

**Performance Evaluation and Real-Time Adaptation:**
Evaluating our strategy's impact on server response times and overall system efficiency was a challenging yet fruitful aspect of the project. Establishing a baseline for comparison was key to understanding the effectiveness of our approach. Adapting to real-time fluctuations in web traffic underscored the necessity for a responsive and adaptable strategy, affirming the importance of a self-adaptive system in dynamic environments.

This project has been a valuable learning experience for our team. It highlighted the potential and complexities of integrating advanced machine learning into real-world, self-adaptive systems, offering insights into both the technological advancements and the practical challenges of such endeavors.

# DIVISION OF WORK

| TEAM MEMBER | RESPONSIBILITIES |
|---|---|
| [redacted] | ● Project Architecture Diagrams<br>● MAPE-K analysis validation<br>● Documentation |
| [redacted] | ● Contextual Bandit algorithm strategy implementation<br>● SDG Regressor algorithm implementation<br>● Visualization of results from different algorithms<br>● Running and debugging the models and strategies on Mac |
| [redacted] | ● Data Collection from EWS simulation<br>● Integration of models into MAPE-K loop<br>● Evaluation of different adaptation strategies<br>● Running and debugging the strategies on Ubuntu (Linux) |
| [redacted] | ● Feature Engineering.<br>● Alternate Implementations with different models and reward functions - Experimenting and finetuning models.<br>● Documentation |