Root-Causing and Event Identification Through Sensor Data

Simon dos Reis Spedsbjerg : sispe20@student.sdu.dk

Advisor: Aslak Johansen : asjo@mmmi.sdu.dk

June 2, 2025

The Maersk Mc-Kinney Moeller Institute

University of Southern Denmark

Contents

1 Abstract

Machine Learning (ML) techniques have become the predominant choice for predictive systems in software engineering. The problem arises as these models often lacks transparency. Making it difficult for developers and engineers to verify the result and ensure that the ML model reflects the real world.

This project introduces a novel software design pattern, **Relational Event Prediction System (REPS)**, which seeks to address interpretability concerns by modelling event relationships through a graph structured system.

REPS defines a prediction system composed of independent nodes that retrieves values from parent nodes. These nodes are configured using JSON with a set of unique keywords. Each node represent either a **Sensor** or **Event**, an Event can contain a user defined expression using C# compiled in run-time or choose an existing implementation of a ML model, such as Support Vector Machines and Random Forest. This means REPS is flexible in its use-case areas.

The implementation demonstrates that REPS is a low-complexity system requiring only 30 configuration keywords, while remaining expressive enough to support a variety of use cases. Runtime performance measurements indicate minimal CPU usage due to parallel node execution, but memory consumption remains relatively high. A simulator was also developed for testing and verification purposes.

This project concludes that while REPS is promising alternative to traditional ML-based systems, further work and testing is needed to determine to what extend REPS can compete.

2 Introduction

The project is very exploratory in nature, it was planned to be a strategy for leveraging graph structure to make determinations, it has since evolved into development of a new software pattern that offers greater value than the original plan in terms of broader applicability. This project attempts to determine best way to design the pattern, it remains within same topic and uses graph structure, REPS introduces a structured approach using a set of rules and relationships between nodes to make determinations. The focus of this thesis is to explore the design space for such a pattern rather than to directly compare it to existing solutions, as optimization and performance benchmarking are not the primary objectives.

This article is structured to follow the chronological progression of the project. After the related work section, the direction of the project shifts noticeably—from evaluating REPS as a competitor to existing predictive systems, to assessing its role as a distinct and interpretable software pattern.

Modern Machine Learning (ML) systems often suffer from a lack of interpretability. Developers cannot always explain or verify how a model makes its determinations. This makes it challenging for developers to understand how the system operates internally, verify the results, or ensure the verifiability of the outcomes. This black box nature introduces a barrier to trust and understanding in their software's determinations.

To address these challenges, I present REPS, a graph-based real-time analysis tool which developers themselves can design to fit their use-case while making use of many state-of-the-art machine learning models. REPS operates by processing input data, which will be either sensor or event data which will pass through nodes called "event nodes" which generates event data. Each node processes inputs from one or more sources and generates a result that can be used on one or multiple downstream nodes. This system enables real-time decision-making across interconnected components while still allowing the developer to know how it operates.

This project aims to determine whether there are any viable use cases for REPS and what advantages and disadvantages it may have.

2.1 Problem

Research Questions:

- RQ0: How should a REPS be constructed?
- RQ1: What advantages and disadvantages does REPS?

Research Question mapping:

- RQ0: The construction principles of REPS can be derived from the practical experience of designing and implementing the system. This question aims to guide future implementations of similar systems by highlighting architectural decisions, trade-offs, and lessons learned throughout the development process.
- RQ1: For REPS to be a viable candidate in real-world environments, it must offer value over existing solutions. Therefore, this question involves a comparative evaluation in terms of CPU usage, memory consumption, configurability, ease of understanding, and predictive capabilities.

3 Context

For this project, some definitions is set to achieve best understanding of the project.

- **Event**, an event is the change of the state of an object, it has one or more inputs and only one output.
- **Model**, a representation of a system or process which can be formed mathematically with the goal of making interpretation of input data.
- **Node**, a node has a output value and a process.
- **Sensor Node**, an egde node where sensor data is received,
- **Event Node**, a node containing a model which it uses to determine an ongoing event.
- **Value**, a value is produced by a sensor or an event.
- **Trigger** An event is triggered when the change of state occurs. When an event is triggered, it emits a value indicating its state, this can varies between a boolean, triggered or stable, or it can vary in severity.
- **ML Model**, A model containing a some form of machine learning implementation to interpret data.
- **Simple Model**, A model which is defined by a function written some form of programming language, in this project, C# is used. It is used to interpret data in scenarios which a custom set of instructions is needed.

REPS is a new software pattern, it is based on a graph structure. This article attempts to develop an implementation of the pattern. From this development an attempt will be made to discover advantages and disadvantages with REPS, outline best practices and common pitfalls in similar solution development, and help define what the REPS pattern should and should not include.

The REPS' nodes includes event nodes and sensor nodes, where event nodes has a model. The model can be a ML model or a specified logical statement. Event nodes job is to determine whether an event is ongoing or is going to

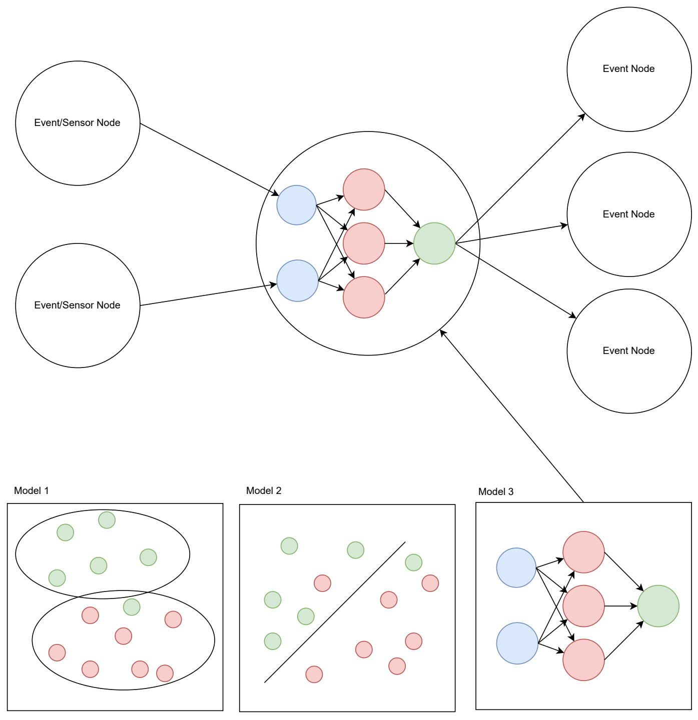

Figure 1: An Eventnode containing a ML model

happen. They give an output to help other event nodes further down in a event graph. Sensor nodes job is to retrieve data from outside the system.

4 Related Work

As this project is exploratory, a literature review is conducted to identify Existing solutions and Machine learning models used. The aim is to understand prior approaches and use this insight to inform the project's direction.

A term mapping is first created to identify relevant search terms. This is done through brainstorming and synonym lookup. Each term is checked for contextual relevance using a synonym dictionary, and valid alternatives are added to the mapping.

Term	Synonym
Event	Act, Action, Case, Circumstance, Crisis, Development, Episode, Experience, Incident, Occasion, Situation, Advent, Calamity, Catastrophe, Concuncture, Emergency, Exploit, Occurrence, phenomenon
Prediction	Forecasting, Indicate, Prognostication, Anticipation
State	Case, Element, Position, Status
Change	Adjustment, development, revision, mutation, correction, transmutation
Value	Content
Rule	Decree, Guideline, Law, Regulation, Ruling, Statute, Criterion, Dictum, Guide, Model, Prescription, Precept, Policy
Trigger	Cause, Activator, Set off, Give rise to, elicit

The first search engine used is Scopus, chosen for its broad coverage across multiple publishers, reducing bias from any single publisher's standards. To ensure the use of recent state-of-the-art research, articles published before 2015 are excluded. Using this constraint and the defined synonym mapping, the following query is applied:

TITLE-ABS-KEY ((event OR act OR action OR case OR circumstance OR crisis OR development OR episode OR experience OR incident OR occasion OR situation OR advent OR calamity OR catastrophe OR concuncture OR emergency OR exploit OR occurrence OR phenomenon) AND (prediction OR forecasting OR indicate OR prognostication OR anticipation)) AND PUBYEAR > 2014 AND PUBYEAR < 2026 AND PUBYEAR > 2014 AND PUBYEAR < 2024 AND (LIMIT-TO (DOCTYPE , "cp") OR LIMIT-TO (DOCTYPE , "ar")) AND (LIMIT-TO (SUBJAREA , "COMP") OR LIMIT-TO (SUBJAREA , "MATH")) AND (LIMIT-TO (LANGUAGE , "English"))

The initial search yields 172,968 articles, which are sorted by citation count to prioritize quality. To maintain source diversity, only a subset of the top 200 results is considered, limiting overrepresentation from a single search engine so to allow time for other search engines. From this set, 32 articles are selected based on title relevance to REPS and their potential to inform the project's development.

The selected articles were published across the following sources:

- ACM : 7,
- Springer Link : 3
- IEEE Xplore : 12
- Oxford Academic : 1
- Nature Biomedical Engineering : 2
- MIT : 1
- MDPI : 1
- ACL Anthology : 1
- Science Direct : 3

The next search engine used is Forskningsportalen, a Danish platform for academic research that includes publications in both Danish and English. It was chosen to reduce duplication of articles found in other databases, as the majority of the already selected articles derives from two search engines and either one of those selected articles may reappear. The query used is the same as for the Scopus search, with adjustments to match Forskningsportalen's syntax:

(event OR act OR action OR case OR circumstance OR crisis OR development

OR episode OR experience OR incident OR occasion OR situation OR advent OR calamity OR catastrophe OR concuncture OR emergency OR exploit OR occurrence OR phenomenon) AND (prediction OR forecasting OR indicate OR prognostication OR anticipation) AND Publication Types=(Journal Article OR Conference Paper OR Thesis PhD) AND Publication Audiences=(Scientific) AND Publication Status=(Published) AND Review Status=(Peer Reviewed) AND Language=(English OR Danish) AND Open Access=(Open Access) AND DK Main Research Areas=(Science/Technology) AND Years=(2024 OR 2023 OR 2022 OR 2021 OR 2020 OR 2019 OR 2018 OR 2017 OR 2016 OR 2015 OR 2014)

The search returned 7,509 results. To reduce the scope, only the 100 most cited articles were considered. Using the same method as the Scopus search, 8 articles were selected based on their titles. After review, 4 were deemed relevant to this project.

In total, 40 articles were reviewed, with 17 selected based on quality. Selection criteria included data and result credibility, does the result align with their data and or experiment, as well as consistency between the abstract and the papers content, does the abstract state something different from the papers content.

Two major concerns are identified across the selected articles: prediction challenges (noted in 9 articles) and feature reduction (noted in 6 articles).

As stated, a record of what ML models used in the reviewed papers would be made for two reasons: to provide a reference set for potential comparison with the final product, and to inform model selection if ML is incorporated into the system. Frequently used models include:

- Neural Networks: 12 (including Recurrent Neural Networks: 9, and Gated Recurrent Units: 7)
- Support Vector Machines (Support Vector Machine (SVM)): 7
- Bayesian Models: 5
- Random Forest: 5
- Nearest Neighbour: 5

4.1 Outcome

The review aims to identify high-value directions and avoid unproductive paths during the analysis phase of software development. Most articles do not explicitly describe the challenges encountered. As a result, potential pitfalls must be inferred by noting which methods or solutions are consistently omitted. Solutions that are rarely mentioned despite being theoretically viable are likely avoided due to having practical problems. Conversely, frequently referenced methods are considered more reliable and are prioritized in this project.

Wu et al.[3] use a graph-based structure for time series forecasting, its idea is very similar to this projects, using graph based relationships to make determinations, their solution takes a different approach, it transforms multivariate time series data into a graph which can be used in a graph neural network, The graph structures nodes derives from variables from multivariate time series data. In relation to this project it shows one approach to the same problem of defining a relation between nodes to predict events. If needed to incorporate automatic graph generation for REPS, taking some of their ideas and transforming it to fit better to that use case could. Incorporating their ideas into REPS could prove valuable if there is time for it.

Ribeiro et al.[2] Propose a technique to explain how the model came to a prediction which can help the end user in trusting the prediction. REPS gains the trust by making the user construct their own system, while if they implement a ML in the traditional sense, they will have to trust it based on testing. Ribeiro et al. proposal of a method to interpret the model afterwards gives the user trust in the model and the determination. This could also be valuable to explain which node had large impact on determining the outcome in the REPS.

These articles fall within the domain targeted by the proposed pattern.

In the early stages of the project, it was considered that an event node could include a ML model. A review of the selected literature where all sources incorporate ML to some extent for prediction indicates a clear trend. Based on this, the project shifts from possibly including ML to should have it as a component.

The literature also informs the approach to user interaction. Ribeiro et al. [2] emphasize user trust through model explain-ability, while Wu et al. [3] contribute structural insights. Based on these, a publish-subscribe pattern will be used for communication between REPS and external components as this seems like best solution to incorporate to communicate the state of the system and nodes.

5 Development

This section covers the development of the product alongside the software engineering methods and tactics applied. The process is structured into three phases: analysis, design, and implementation.

5.1 Methods

This section outlines the methods applied during the development phase of the project.

5.1.1 Time & Cost

Accurate time and cost estimation is a persistent challenge in software development. Common methods include:

- Expert methods, such as Delphi Technique, Planning Poker, Analogous Estimating, and Expert Judgement.
- Algorithmic Methods, such as COCOMO, Function Point Analysis, and Use Case Points.

Each method has strengths depending on project scope, team size, and experience. For this project, Expert Judgment is used, as the technologies involved are familiar. This allows for reliable estimates of feature development time.

For larger or less technology specific experienced teams, a different method may be more suitable, as Expert Judgment relies heavily on prior experience.

Feature prioritisation is guided by evaluating dependencies and the user value each feature provides. A cost-benefit analysis is conducted at the start of each iteration to ensure alignment with project scope.

5.1.2 Development model

Software development is often guided by two main methodologies: Waterfall and Agile. Waterfall is suitable when requirements are unlikely to change, such as in medical software. Waterfall often follows these steps:

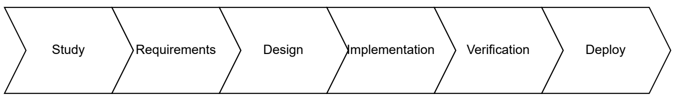

Figure 2: Waterfall Workflow

Agile is more adaptive and supports frequent changes in requirements, which makes it valuable for this project due to its exploratory nature. The stages in agile is often circular and allows for going back and forth in the stages.

This project applies some SCRUM practices, with iterations limited to two weeks. After each iteration, the project state is reviewed to support requirement adjustments. The first day of each iteration is reserved for documentation and

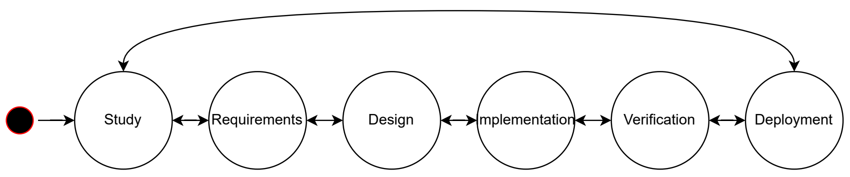

Figure 3: Iterative Workflow

reflection on the previous cycle. The class diagram is updated regularly to maintain a clear overview of the system.

Maintaining a high-level overview enables early identification of flawed implementations, allowing timely corrections before issues escalate.

5.2 Development Tools

Tools that monitor memory and CPU usage can help identify problems or bottlenecks early in the development process. Additional tools are developed and used to accelerate testing and experimentation. A custom GUI was created to streamline testing by enabling easy deployment of a varied set of test cases. Docker is used for experiment deployment, as it also provides the ability to track CPU, memory, and network usage.

5.3 Analysis

To determine the optimal design based on insights from the research phase, several analytical methods are applied. First, features are separated into those likely to provide value to the end user and those considered less relevant. This is done through brainstorming, as other methods are less effective in producing satisfactory results according to the developers experience.

5.3.1 Potential Features

- Modability during run-time : The idea here being that a user would never need to have any downtime even when they only run one instance of REPS. This would lean towards a component based design of REPS, but could be implemented through other means.
- Adding or removing nodes during run-time : This comes with the same general idea which could reduce down time when using only one instance.
- Hybridize with other predictive models : Someone could insert a ML model while still keeping memory usage low, this would allow REPS to extremely complex in is design and cover a large use case. It can be imagined that REPS already is going to require a large usage of memory, therefore models which is memory demanding such as K's Nearest Neighbour (KNN) will be unusable in a REPS.

5.3.2 Requirements

A MoSCoW analysis is conducted to determine the prioritisation of requirements. The outcome of this analysis informs the selection of technologies used in the project, as choices are guided by each technology's ability to support the identified requirements.
A clear project scope is defined early to maximize the value of collected experience and data from this project. Therefore a MoSCoW analysis is used to prioritize features.
Functional Requirements:

Requirement ID	Requirement	Prioritization Level	Description	Reason
FR0	Individual process in each 'Event Node'	Must have	An 'Event Node' is the main process for making predictions.	These processes are essential for making predictions.
FR1	Ability to communicate with external services not part of REPS	Must have	External systems must be able to receive values from REPS to make decisions based on REPS' observations.	For external systems to react to REPS' state changes, REPS must share its data. Without this, REPS serves no purpose.
FR2	Easily add new 'Event Nodes'	Could have	Using XML, JSON, or another format, new nodes could be added to the REPS.	May accelerate testing, though not required for system functionality.
FR3	Quickly initialise REPS using a configuration file	Could have	Allows quicker deployment and enables sharing of REPS configurations.	It could speed up testing, but it is not a requirement for the system to function
FR4	Support for ML models in 'Event Nodes'	Should have	Would expand REPS' use cases, enabling companies to integrate or develop models tailored to specific problems.	Highly useful for this project but not essential for evaluating system viability.
FR5	Event Detection Capability	Must have	Detects relevant events occurring within the system.	Core functionality of the system.
FR6	Graphical user-interface	Wont have	A user interface would simplify REPS setup for developers.	Deemed too time-consuming and not essential for evaluating system viability.
FR7	Automatic REPS creation	Wont have	Could theoretically enable automatic REPS generation from system documentation.	Considered a separate research topic and beyond the scope of this project.

Wont have means I do not plan to implement such feature at the moment, but is included as this could have value for future work.
Non-Functional Requirements:

Requirement ID	Requirement	Prioritization Level	Description	Reason
NF0	Should not have a high complexity	Should have	Complexity refers to how easily a developer can understand the system.	A key advantage is that the developer should be able to understand how a decision is made.
NF1	Restricted Memory usage	Could have	There is a risk of high resource usage.	The system may suffer from memory issues on lower-end or smaller computers.
NF2	Easy to communicate with	Should have	It should be easy to retrieve necessary values from REPS.	If this is not fulfilled, using REPS would become difficult, significantly reducing its usefulness.

5.3.3 Technology Choices

Based on these requirements, the following technologies, frameworks, and patterns are considered necessary:

- **Publish-Subscribe pattern, MQTT (FR1, FR2, FR3, NF2):**
  The publish-subscribe pattern best suits the communication needs. Its modifiability and adaptability align well with the communication requirements (FR1) while supporting easy extension (FR2 & FR3). Message Queuing Telemetry Transport (MQTT) supports all necessary features and is therefore picked due to its simplicity and light weight.

- **JSON interpreter for saving (FR2, FR3):**
  A relational database does not suit REPS's structure. JavaScript Object Notation (JSON) or XML provide the flexibility required. There is no need for the performance benefits of a relational database. JSON is preferred, as I have more experience implementing it, whereas XML is less familiar. AVRO will not be used as it will require more development time than JSON.

- **C#:**
  I have the most experience with C#. While it may be slower than languages like C/C++, it is sufficiently fast for this project. Its object-oriented nature suits REPS well. A drawback is the limited availability of ML libraries compared to Python, which may slow the implementation of FR4 but will enable faster development of other features.

- **Roslyn:**
  Roslyn is a .NET compiler platform which allows for turning the C# compiler into a programmable platform. Meaning I can compile code during run time. As anything compiled during runtime, it will require more computational power, an increase in CPU and Memory usages is expected. Roslyn allows for the simplest form of making determinations in a node and is therefore included even though it may have Memory usage problems.

- **ML models (FR4):**
  A simple model will use Roslyn to make determinations, but as the related work (section 4) determined, ML models frequently used to make determinations and supporting that will most likely bring the highest value. Through this project, the following ML models in same following order will be implemented. SVM, easy to implement and is the second most used model in the related work. Random Forest (RF) due to it being very different from SVM, allowing to determine the range of models which can be used in REPS. Neural Networks requires more implementation than some of the other 2, while it is the most used according to the related work, it also requires more work, and as the implementation of models is not the focus of this project, it can be put in the last spot in the queue for implementation.

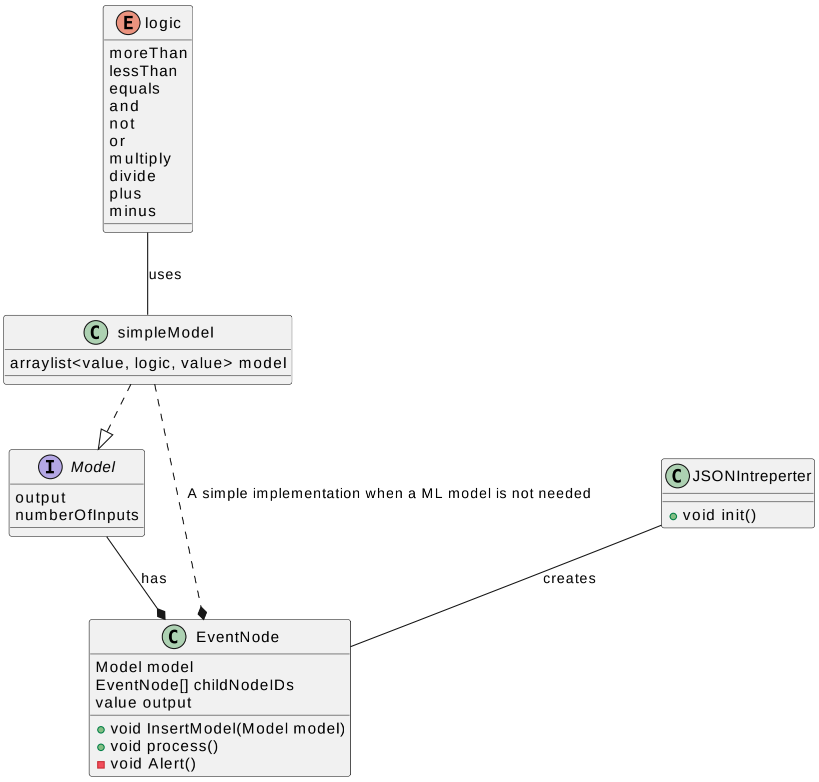

Figure 4: First Iteration Design Diagram

Based on the diagrams in Figure 4, RabbitMQ will be used for MQTT as it meets all project requirements and supports C# implementation. It offers classic MQTT features and queue prioritisation, which may be essential in crisis situations. The number of priority levels should be limited, as increasing them demands more CPU resources. Additionally, messages set to expire might remain longer than expected, since RabbitMQ expires messages from the head of the queue. However, this should not pose a problem provided the end user does not send excessive messages.

In C#, the output must be defined as the `object` type to allow broad usage of different types. While the `dynamic` type is an alternative, it introduces runtime casting, which increases the risk of errors. The `var` keyword is not suitable, as it resolves types at compile time. By using `object`, I must explicitly define a set of permitted casting types. This reduces flexibility but significantly lowers the risk of runtime errors.

I will use C#'s Roslyn to interpret functions at runtime. This enables full customisation of logic, including mathematical operations and conditional statements.

For JSON parsing, I will be using Newtonsoft, a C# framework for interpreting JSON. This choice is based on its ease of use, extensive documentation, and the fact that it is the most downloaded NuGet package[1], which suggests that if any issues arise, solutions are likely to be found on Stack Overflow or similar platforms.

The target of the project is to have and be able to create graphs, a worry then arrives that the implementation is has a tree like structure that is pretends to be a graph, i.e. that it is not possible to create circular dependencies but all nodes descend from a set of root sensor nodes. Therefore there is a need to remove dependencies from other nodes, that even though the parent node is not finished processing, the child node will just have to use the old output value. This ensures that a circular dependency does not cause an infinite loop and actually allows for it to be a graph.

5.4 Design

Based on the analysis (section 5.3), a set of technology choices (section 5.3.3) could be stated which is needed to achieve the requirements (section 5.3.2), and on the technology choices, a more precise design is made to align with the technology.

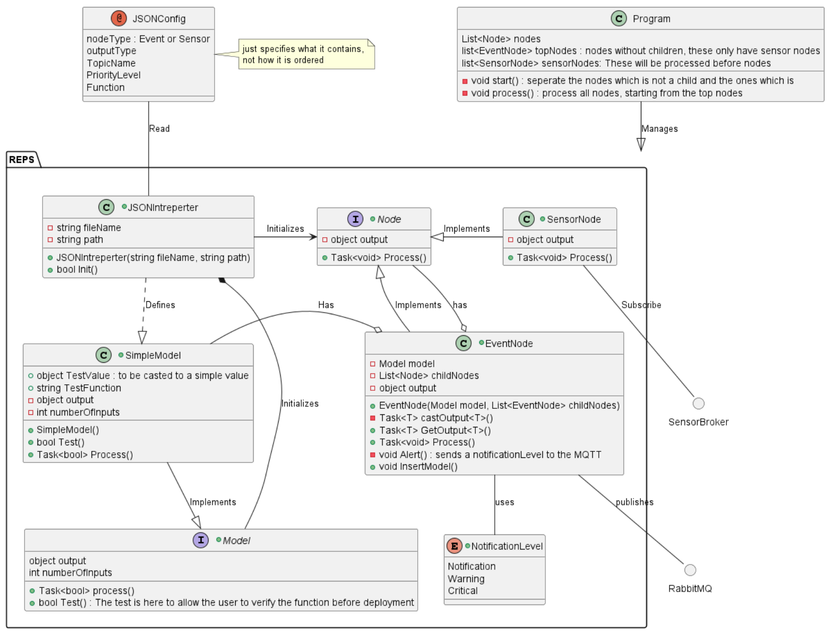

Figure 5: First Design

As the project furthers and new iterations of the project is completed, the class diagram will be updated with the final diagram (Figure: 7), this diagram is just to show the size of the final program when more of the planned features has been developed, as it is too bige to properly represent on paper. It mostly remains in the same theme as figure 5: The REPS is split up in 2 parts, the system itself and then a GUI.

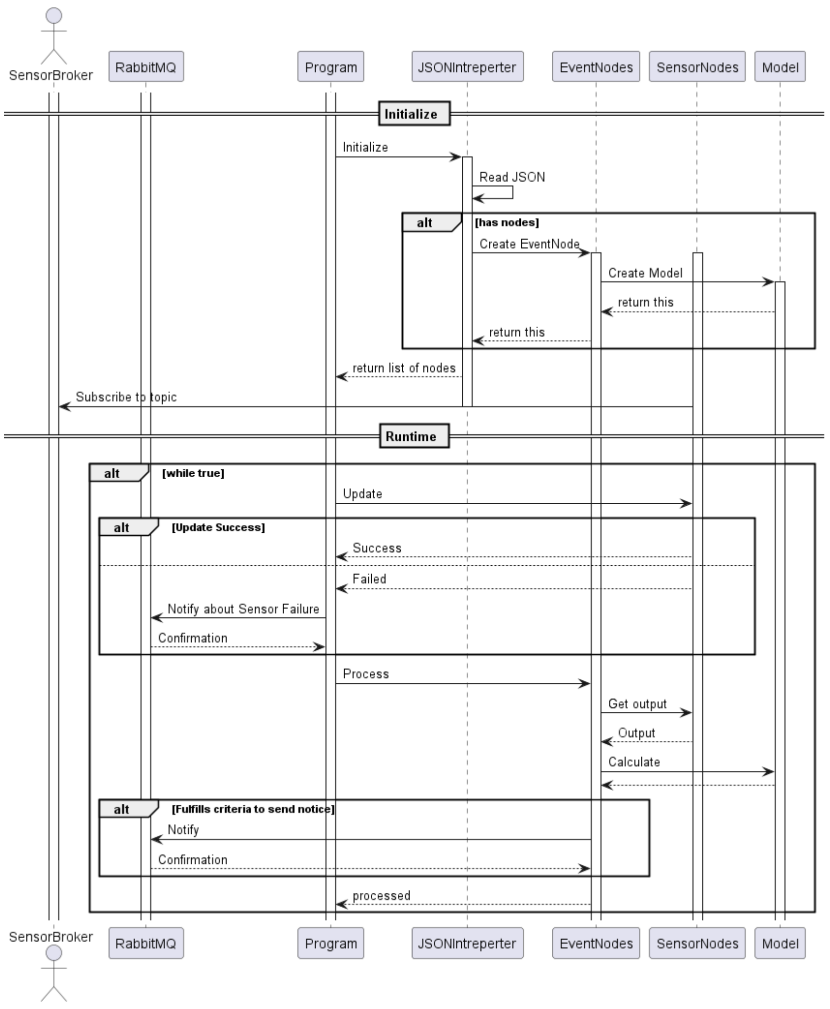

Figure 6: Sequence Diagram

Logical Insurance
To ensure the correctness of the system's logic, UPPAAL, a model-checking tool is used to validate the system's logic before implementation. This reduces the risk of introducing logical errors early in development.

5.5 REPS Implementation

REPS begins by loading a configuration file in JSON format. JSON was chosen due to its ease of modification during testing, as well as its compatibility with other systems such as the GUI. The configuration defines the various nodes in the system, with each node represented by a struct containing all relevant data.

A sensor node receives input either from a physical sensor or via another communication method. It was determined that the nodes should follow a pub-

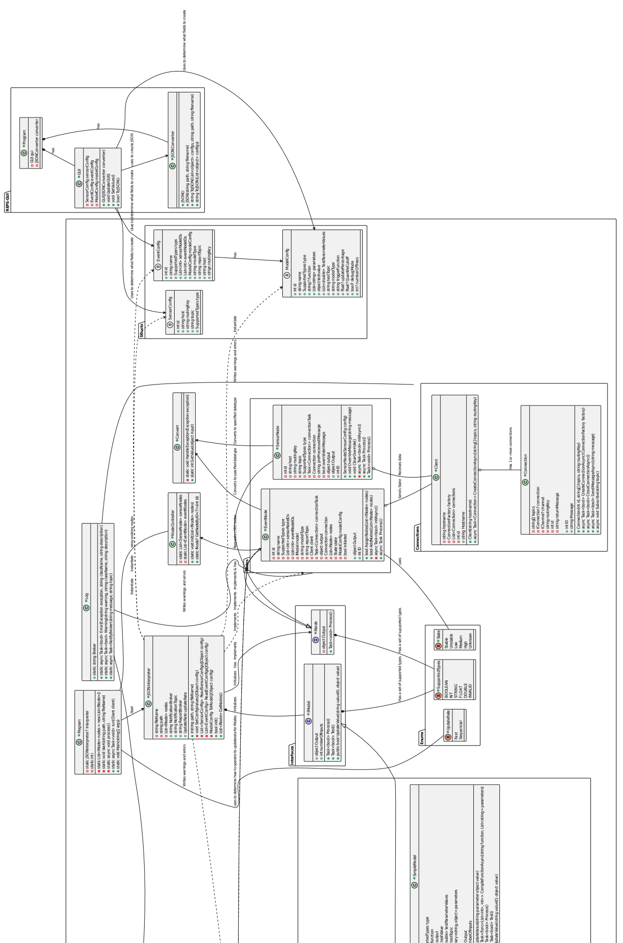
Figure 7: Final ClassDiagram

lish–subscribe pattern, as this suits their dynamic nature. RabbitMQ was used for all communication, primarily due to its support for priority queuing.

An event node retrieves data from sensor nodes and applies its model to the input. The model's output is then made available to other nodes. Each event node also includes a trigger function, which is used to assess the severity of the ongoing event.

Each node connects to the MQTT broker using a specific topic, which it either publishes to or subscribes to for receiving data. An event node publishes to the broker based on the severity of the event and user-defined preferences.

5.5.1 Models

Simple models use C# Roslyn's compiler to compile code at runtime, allowing for custom functions to make determinations. By using Roslyn, full customisation is possible without the need to implement a separate interpreter. However, this increases complexity for the user, as they must write functions using C# syntax, and it introduces a risk of arbitrary code execution.

Each Event Node contains a model which it uses to determine an output. The complexity of the model can vary; a simple model, can take a mathematical function written in C# syntax and generate an output based on it.

The following models have been implemented based on related work(section 4).

- SimpleModel : A model intended for users to define their own logic using a simple expression. While it is Turing complete, it is not suited for complex algorithms due to higher memory consumption.
- TimedModel : Used to evaluate actions over time. One use case is calculating the average number of new unique entries within a time window. It acts more as a supporting model to help other models making their determinations.
- AdaptivModel : Adjusts its trigger threshold dynamically to improve prediction accuracy. Useful when the optimal threshold is unknown or unreliable.
- MachineLearningModel : An abstract class containing shared methods for all ML model implementations.
- SupportVectorMachineModel : A model that implements the Support Vector Machine.
- RandomForestModel : A model that implements the Random Forest.

A Simple models implementation has to compile the trigger and function

```
protected virtual async Task<Func<object[], object?>> CompileFunctionAsync(string function, string[] parameters) {
  string guid = Guid.NewGuid().ToString("N");
  string code = $@"
  using System;

  public class DynamicFunction_{guid} {{
      public static {REPS.Convert.GetStringType(type)} Compute_{guid}({string.Join(", ", parameters.Select(p => $"object {p}"))}) {{
          {function};
      }}
  }}";
  MethodInfo method = MethodFunctionConstructor(code, guid);
  return args => {
    object[] argumentArray = args.Select(x => (object)x).ToArray();
    object? invokation = null;
    try {
      invokation = method.Invoke(null, argumentArray);
    }
    catch(Exception e) {
      ...
    }
    return invokation;
  };
}

```

Code Snippet 1: SimpleMode.cs

The method must be constructed each time it is processed. Since the model needs to be dynamic, accepting any code, input, and output, it requires compilation every time. This is a requirement of the Roslyn framework. Compiling at runtime also enables changing the model dynamically.

An Adaptive model functions similarly to the Simple model, but it maintains an equilibrium value which it updates during runtime.

```
protected virtual void UpdateEqulibrium<T>((object, TimeOnly)[] values) {
  if(typeof(T) == typeof(int)) {
    int cutOffPoint = (int)(this.values.Count * updatePercentage);
    TimeOnly timeCutoff = this.values[cutOffPoint].Item2;
    (object, TimeOnly)[]? toBeReplaced = ((object, TimeOnly)[])this.values.Where(data => data.Item2 > timeCutoff);
    foreach((object, TimeOnly) value in this.values) {
      if(toBeReplaced.Contains(value)) {
        this.values.Remove(value);
      }
    }
    this.values.AddRange(values);
    this.triggerLowerEqulibrium = this.values[(int)(this.values.Count * QuantileCutoff)].Item1;
    this.triggerUpperEqulibrium = this.values[(int)(this.values.Count * (1 - QuantileCutoff))].Item1;
  }
  ...
}

```

Code Snippet 2: AdaptivModel.cs

It updates the trigger limit, which determines when an event is considered triggered.

A Machine Learning (ML) model is an abstract class and is also regarded as an adaptive model. However, in the current implementation, it does not utilise the inherited functionality. The concept is that ML models can also adapt as new data becomes available. The ML model defines two methods which are not implemented as it is expected to be implemented in class' which inherent from this.

ML models cover a wider range of use cases and are easier to implement from a user's perspective. Both SVM and RF have been implemented to test their viability. The SVM model uses Microsoft.ML, which provides much of the implementation needed for different models. The SVM requires only a path to the training data; the data is then used to train the model.

```
public override object Predict(object data) {
  Prediction prediction = predictionEngine.Predict((Data)data);
  Console.WriteLine($"Prediction: {prediction.PredictedLabel} | Score: {
    prediction.Score}");
  lastPrediction = prediction.PredictedLabel;
  return prediction.Score;
}

public override bool Train(object[] inputData) {
  try {
    List<Data> trainingData = new List<Data>();
    foreach(object input in inputData) {
      trainingData.Add((Data)input);
    }
    var vectorSize = trainingData[0].Features.Length;
    Console.WriteLine(trainingData[0].Features.GetType());
    trainingData = trainingData.Where(x => x.Features.Length ==
    vectorSize).ToList();
    IDataView? data = context.Data.LoadFromEnumerable(trainingData);

    var pipeline = context.Transforms.NormalizeMinMax("Features").Append
    (context.Transforms.NormalizeMinMax("Features")).Append(context.
    BinaryClassification.Trainers.LdSvm(labelColumnName: "Label",
    featureColumnName: "Features"));

    Console.WriteLine($"Training...");
    this.model = pipeline.Fit(data);
    this.predictionEngine = context.Model.CreatePredictionEngine<Data,
    Prediction>(model);
    return true;
  }
...
}

```

Code Snippet 3: SupportVectorMachineModel.cs

The models use the score to make a prediction; the higher the confidence of the model in its outcome, the greater the severity assigned.

RF follows very similar setup except you can need to define number of trees. The models follows the class digram defined in figure 8

5.5.2 Trigger

A trigger is implemented similarly to a simple models code compilation; however, it is more limited in what it can return. This restriction exists because such flexibility is unnecessary and it reduces the risk of arbitrary code execution.

5.5.3 JSON Reading

The JSON Interpreter's role is to read a configuration file that an end user can modify to launch the REPS. The intention is to keep the user's implementation of a REPS system simple. In this sense, the JSON reading acts as a Doman-Specific Language (DSL), where maintaining low complexity is essential for this project. Therefore, the JSON reader should include only a limited number of keywords.

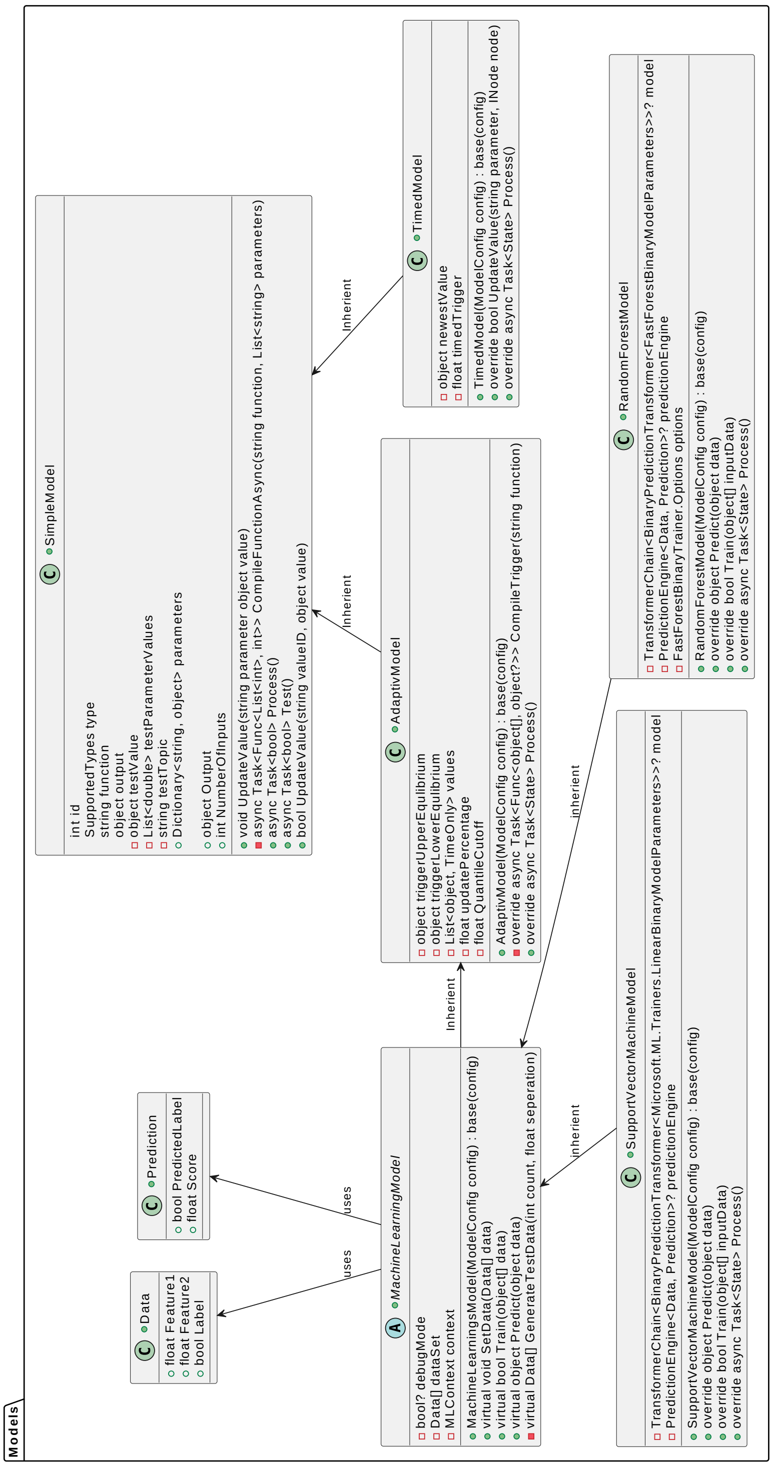
Models
SimpleModel
int id
SupportedTypes type
string function
object output
object testValue
List<double> testParameterValues
string testTopic
Dictionary<string, object> parameters
object Output
int NumberOfInputs
void UpdateValue(string parameter object value)
async Task<Func<List<int>, int>> CompileFunctionAsync(string function, List<string> parameters)
async Task<bool> Process()
async Task<bool> Test()
bool UpdateValue(string valueID, object value)
TimedModel
object newestValue
float timedTrigger
TimedModel(ModelConfig config) : base(config)
override bool UpdateValue(string parameter, INode node)
override async Task<State> Process()
AdaptivModel
object triggerUpperEquilibrium
object triggerLowerEquilibrium
List<object, TimeOnly> values
float updatePercentage
float QuantileCutoff
AdaptivModel(ModelConfig config) : base(config)
override async Task<Func<object[], object?>> CompileTrigger(string function)
override async Task<State> Process()
RandomForestModel
TransformerChain<BinaryPredictionTransformer<FastForestBinaryModelParameters>>? model
PredictionEngine<Data, Prediction>? predictionEngine
FastForestBinaryTrainer.Options options
RandomForestModel(ModelConfig config) : base(config)
override object Predict(object data)
override bool Train(object[] inputData)
override async Task<State> Process()
MachineLearningModel
bool? debugMode
Data[] dataSet
MLContext context
MachineLearningsModel(ModelConfig config) : base(config)
virtual void SetData(Data[] data)
virtual bool Train(object[] data)
virtual object Predict(object data)
virtual Data[] GenerateTestData(int count, float seperation)
SupportVectorMachineModel
TransformerChain<BinaryPredictionTransformer<Microsoft.ML.Trainers.LinearBinaryModelParameters>>? model
PredictionEngine<Data, Prediction>? predictionEngine
SupportVectorMachineModel(ModelConfig config) : base(config)
override object Predict(object data)
override bool Train(object[] inputData)
override async Task<State> Process()
Data
float Feature1
float Feature2
bool Label
Prediction
bool PredictedLabel
float Score
uses
uses
Inherient
Inherient
Inherient
inherient
inherient

Figure 8: Current Implemented models

```
private List<SensorConfig> ReadSensorConfigs(JObject config) {
  ...
  foreach(JObject sensorNode in sensorNodes) {
    try {
      SensorConfig sensorConfig = new SensorConfig();
      sensorConfig.id = (int)sensorNode.GetValue("ID");
      sensorConfig.host = sensorNode.GetValue("Host").ToString();
      sensorConfig.name = sensorNode.GetValue("Name").ToString();
      sensorConfig.type = DetermineSupportedType(sensorNode);
      if(sensorConfig.type == SupportedTypes.STRING) {
        ...
      }
      sensorConfig.topic = sensorNode.GetValue("Topic").ToString();
      sensorConfig.routingKey = sensorNode.GetValue("RoutingKey").
    ToString();
      sensorConfigs.Add(sensorConfig);
    }
    ...
  }
  return sensorConfigs;
}

```

Code Snippet 4: Reading Sensor Configs

Reading configuration files is implemented as shown in Code Snippet 4, where the code interprets specific values and stores them in a struct. This struct is then used to instantiate the models.

```
public struct SensorConfig {
  public int id;
  public string name;
  public string host;
  public string routingKey;
  public string topic;
  public SupportedTypes type;
  public bool? isUsername;
}

```

Code Snippet 5: SensorConfig

Implementing all of the configurations in the JSON reader, from which we can determine the amount of keywords needed implement the current solution. There is a total of 30 keywords, a list can be found in appendix 9.1.

With the current implementation of the JSON reader, a user can implement REPS. For example, a simple model that takes two sensors and adds their values can be defined as shown in Code Snippet 6.

```
{
  "BrokerInfo": {
    "NotificationBroker": "localhost",
    "NotificationTopic": "notify",
    "ReportBroker": "localhost"
  },
  "SensorNodes": [{
    "ID": 0,
    "Name": "first",
    "SupportedType": "INT",
    "Host": "localhost",
    "RoutingKey": 0,
    "Topic": "first"
  },{
    "ID": 1,
    "Name": "second",
    "SupportedType": "INT",
    "Host": "localhost",
    "RoutingKey": 0,
    "Topic": "second"
  }],
  "EventNodes": [{
    "ID": 0,
    "Name": "Name",
    "SensorNodes": [ 0, 1 ],
    "EventNodes": [],
    "ReportTopic": "EventOutput",
    "SupportedType": "INT",
    "Host": "localhost",
    "RoutingKey" :  0,
    "Model": {
      "ID": 0,
      "Type": "Simple",
      "Name": "Name",
      "SupportedType": "INT",
      "Function": "a+b",
      "Parameters": [ "a", "b" ]
    }
  }
]
}
```

Code Snippet 6: A simple model for addition

A more realistic use of REPS is shown in Code Snippet 7. It implements one sensor and three events to determine the average time between bot creations, detect a single bot creation event, and identify an attack involving multiple new bot creations.

```
{"BrokerInfo": {
    "NotificationBroker": "localhost",
    "NotificationTopic": "notify",
    "ReportBroker": "localhost"
  },
  "SensorNodes": [{
    "ID": 0,
    "Name": "AccountCreationSensor",
    "SupportedType": "STRING",
    "Host": "localhost",
    "RoutingKey": 0,
    "Topic": "AccountCreationActivity"
  }],
  "EventNodes": [{
    "ID": 0,
    "Name": "AccountCreationTimerEvent",
    "SensorNodes": [ 0 ],
    "EventNodes": [],
    "ReportTopic": "BotDetection",
    "SupportedType": "FLOAT",
    "Host": "localhost",
    "RoutingKey": 0,
    "Model": {
      "ID": 0,
      "Type": "timed",
      "DebugMode": false,
      "Name": "AccountCreationTimerModel",
      "SupportedType": "FLOAT",
      "TimedTrigger": 0.5,
      "Parameters": ["a"]
  }}, {
    "ID": 1,
    "Name": "BotDeterminerEvent",
    "SensorNodes": [ 2 ],
    "EventNodes": [],
    "ReportTopic": "BotDetection",
    "SupportedType": "INT",
    "Host": "localhost",
    "RoutingKey": 0,
    "Model": {
      "ID": 1,
      "Type": "svm",
      "DebugMode": false,
      "Name": "BotDeterminerModel",
      "SupportedType": "INT",
      "TrainingData": "C:/../TrainingsData.csv"
  }},{
    "ID": 2,
    "Name": "BotCreationAttackDeterminerEvent",
    "SensorNodes": [],
    "EventNodes": [ 0, 1 ],
    "ReportTopic": "BotCreationAttackDeterminer",
    "SupportedType": "INT",
    "Host": "localhost",
    "RoutingKey": 0,
    "Model": {
      "ID": 1,
      "Type": "simple",
      "DebugMode": false,
      "Name": "BotDeterminerModel",
      "SupportedType": "BOOLEAN",
      "Function": "return ((float)a > 0f && (float)b > 0.5)",
      "Parameters": [ "a", "b" ],
      "TriggerFunction": "if((bool)output) return 1; else return 0;"
  }}]}
```

Code Snippet 7: BotCreation Event

5.5.4 Nodes

Any kind of nodes implements an INode

```
namespace REPS.Interfaces {
  public interface INode {
    object Output {
      get;
      set;
    }

    public Task Process();

  }
}

```

Code Snippet 8: INode.cs

The ‘Output’ is where other nodes can access the value determined by the process. The process returns a task, allowing it to run in parallel with other tasks.

```
foreach(INode node in nodes) {
  if(node is EventNode) {
    tasks.Add(node.Process());
  }
}

```

Code Snippet 9: Program.cs

5.5.5 RabbitMQ

The RabbitMQ implementation is divided into two classes: the Client, which manages all connections with the broker, and the Connection class, which handles direct communication with the broker.

The Client has a simple implementation, as the RabbitMQ library already manages most of the background connections.

```
namespace REPS.Connections {
  public class Client {
    private string hostname;
    private ConnectionFactory factory;
    private List<Connection> connections;
    private int id;
    public Client(string hostname) {
      this.hostname = hostname;
      this.factory = new ConnectionFactory { HostName = this.hostname };
      id = 0;
      connections = new List<Connection>();
    }

    public string Hostname { get { return hostname; } }

    public async Task<Connection> CreateConnectionAsync(string[] topics,
     string routingKey) {
      Connection connection = new Connection(id, topics, routingKey);
      id++;
      await connection.CreateConnectionAsync(this.factory);
      connections.Add(connection);
      return connection;
    }
  }
}

```

Code Snippet 10: Client.cs

Once a client has been instantiated, nodes can establish connections. The primary methods used are `Subscribe` and `SendMessage`.

```
public async void Subscribe(string topic) {
  QueueDeclareOk queueDeclareResult = null;
  try {
    queueDeclareResult = await channel.QueueDeclareAsync();
  }
  catch(NullReferenceException ex) {
    ...
    return;
  }
  string queueName = queueDeclareResult.QueueName;
  await channel.QueueBindAsync(queue: queueName, exchange: topic,
    routingKey: routingKey);
  AsyncEventingBasicConsumer consumer = new AsyncEventingBasicConsumer(
    channel);
  consumer.ReceivedAsync += (model, ea) => {
    byte[] body = ea.Body.ToArray();
    string message = Encoding.UTF8.GetString(body);
    this.returnMessage = message;
    return Task.CompletedTask;
  };
  await channel.BasicConsumeAsync(queue: queueName, autoAck: true,
    consumer: consumer);
}

```

Code Snippet 11: Connection.cs

With this implementation sensor nodes can receive data from the broker.

```
private async Task<bool> initAsync() {
  connection = connectionTask.Result;
  connectionTask.Dispose();
  connection.Subscribe(topic);
  return true;
}

```

Code Snippet 12: SensorNode.cs

And it is possible for event nodes to send their determination to the broker.

```
    ...
    this.state = await model.Process();
}

if(state == State.Stable) {
await connection.SendMessageAsync($"EventNode {id}, {name}, reports
    stable condition : Statelevel: {State.Stable}");
}
else {
await connection.SendMessageAsync($"EventNode {id}, {name}, reports
    unstable condition : Statelevel: {state}");
}

```

Code Snippet 13: EventNode.cs

5.5.6 Miscellaneous

In this section, I cover notable points that do not warrant separate sections.

Threading
Each node runs in its own thread, avoiding blocking between nodes. However, faster nodes may read data from others that have not yet finished processing.

Name Extraction
It proved useful to analyse username characteristics within REPS rather than

in a separate service. This was beneficial in the use case shown in Code Snippet 7. It led to the development of an extraction module, which could follow a code injection pattern if extended, enabling broader and more precise use of ML models in REPS.

Supported Types

Supported types are those recognised by REPS. Initially, support for complex types was considered, but this now appears unnecessary as there is no evident demand beyond simple types.

5.6 Simulator

A simulator was developed to mimic website traffic, allowing verification that the REPS implementation functions as intended. While unit tests ensure individual components work correctly, testing the entire system in realistic scenarios often reveals issues not covered by unit tests. The simulator generates requests to a website, which then publishes information about this simulated 'attack' to the RabbitMQ broker. The simulator will have several different type of events

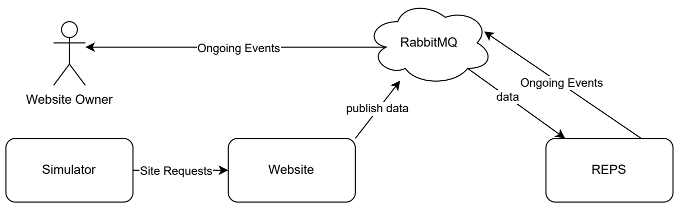

Figure 9: Simulator Task

which can occur such as login attempts and creation of new accounts.

```
class WebSiteLoginEvent : WebSiteActionEvent {
  public WebSiteLoginEvent(string site , int seed , int minimumRequests ,
    int maxmimumRequests) : base(site , seed , minimumRequests ,
    maxmimumRequests , null) {
    this.message = new JObject();
  }

  public override async void Start() {
    for(int j = 0; j < numberOfAttacks; j++) {
      string name = ""; string password = "";
      int length = random.Next(20) + 2;
      for(int i = 0; i < length; i++) {
        name += random.Next(2) == 1 ? letters[random.Next(letters.Length
  )] : letters[random.Next(letters.Length)].ToString().ToUpper();
      }
      for(int i = 0; i < length; i++) {
        password += random.Next(2) == 1 ? letters[random.Next(letters.
    Length)] : letters[random.Next(letters.Length)].ToString().ToUpper
    ();
      }
      this.message["username"] = name;
      this.message["password"] = password;
      await client.PostAsync(site + "/login", new StringContent(message.
    ToString(), Encoding.UTF8, "application/json"));
    }
  }
}

```

Code Snippet 14: Simulator.Event.cs

The simulator also supports saving the data into a csv file, which can be used to train the ML models.

5.7 GUI

The GUI was developed to assist with testing and experimentation. It serves both as a development tool and as part of the final product for the user. The GUI is built using C# Blazor, chosen for its Linux support—which other C# frameworks lack—and my familiarity with it. Using the same language as REPS enables direct use of the config files implemented in REPS (see Section 5.5).

Since the GUI is primarily a development aid, I do not plan to update it alongside REPS. Instead, it dynamically adapts to the config files, easing development and testing. The GUI reduces misconfigurations and speeds up the creation of new configurations by generating JSON files used to launch a REPS instance.

Blazor was also selected for its component-based architecture, allowing easy addition or reuse of pages and components. Alternative frameworks such as WinUI, Windows Forms, and MAUI were considered but dismissed due to either lack of Linux support or my limited experience with them.

Development of the GUI was never completed, as it was not deemed essential for REPS. Work ceased once it became clear the effort would not yield sufficient value.

```
public List<(string, string, Guid, int)> GetTypes<T>() {
  List<(string, string, Guid, int)> types = new(); // I did not use
    dictionary to avoid the risk of macthing keys

  if (typeof(T) == typeof(SensorConfig)) {
    foreach (FieldInfo? field in fieldsTypes[0].GetFields(BindingFlags.
    Public | BindingFlags.Instance)) {
      Guid guid = Guid.NewGuid();
      AddGuidToKey(configNumber, guid);
      types.Add((field.Name, field.FieldType.Name, guid, configNumber));
    }
  }
  else if (typeof(T) == typeof(EventConfig)) {
    foreach (FieldInfo? field in fieldsTypes[1].GetFields(BindingFlags.
    Public | BindingFlags.Instance)) {
      types.Add((field.Name, field.FieldType.Name, Guid.NewGuid(),
    configNumber));
    }
  }
  else if (typeof(T) == typeof(ModelConfig)) {
    foreach (FieldInfo? field in fieldsTypes[2].GetFields(BindingFlags.
    Public | BindingFlags.Instance)) {
      types.Add((field.Name, field.FieldType.Name, Guid.NewGuid(),
    configNumber));
    }
  }
  return types;
}

```

Code Snippet 15: JSON.razor

This method has access to the same struct config types used in REPS, such as Sensor Config [Code Snippet 5]. This allows automatic generation of a table where users can input the relevant information.

5.8 Testing

Testing methodologies in software engineering vary widely, including approaches such as test-driven development, unit testing, and UI testing. These methods help ensure the final product meets its requirements.

For this project, unit testing is important but insufficient to cover the entire system, especially for REPS. As REPS is a supportive system intended to detect issues within its environment, a separate testing system (The Simulator) has been developed to evaluate REPS effectively.

5.9 Iterations

This project has gone through a total of 9 iterations.

- **Iteration 1**: Developed the main feature of the simple model; the event trigger feature is not yet implemented.
- **Iteration 2**: It was decided that a GUI would add significant value. Development began but was left unfinished, as it was deemed not worthwhile to continue.
- **Iteration 3**: The trigger feature and the Minimal Viable Product (MVP) was completed, and simple experiments and testing commenced to assess the project's potential and future direction. Development of the Simulator also started.
- **Iteration 4**: Began addressing technical debt and created the Adaptive model.
- **Iteration 5**: Resolved outstanding issues, including missing features and bugs.
- **Iteration 6**: Added support for ML models and implemented SVM.
- **Iteration 7**: Implemented RF.
- **Iteration 8–9**: Fixed issues and conducted thorough testing of the implementation.

5.10 Research Question 0 : How should a REPS be constructed

A REPS should be constructed based on this project's experience alongside established patterns and software engineering principles. Separation of concerns is inherent to REPS by design. Nodes only retrieve data and do not publish to other nodes, facilitating the easy addition and removal of nodes.

Using a human-readable configuration file allows users to make quick changes during and between runtimes. The pattern's design supports simple setup and interaction, while minimising the learning curve compared to similar solutions.

The best way to implement REPS is with each node being independent, allowing varied implementations. Although this project is implemented as a monolith, there is no reason it could not be implemented in a microservice architecture. Independence enables nodes to perform calculations concurrently without waiting for others to finish. There should be a clear separation between sensor nodes and event nodes, as their concerns differ, and multiple event nodes can utilise the same sensor data. Event nodes should have only one output, ensuring they remain focused on their task.

6 Evaluation

This section evaluates the implementation status and its alignment with initial expectations. It details what has been implemented, to what extent, and assesses the quality and completeness of the implementation.

Additionally, it reviews the features outlined in the MoSCoW analysis, evaluating the degree to which each has been implemented and whether it meets the intended requirements.

6.1 Requirements Evaluation

- **FR0 - Individual process in each 'Event Node':** This works as planned and expected. C# tasks enable nodes to run in parallel as intended.
- **FR1 – Ability to communicate with external services not part of REPS:** Implemented using RabbitMQ, which successfully enables service-to-service communication. The only missing feature is queue prioritization, which was deferred in favour of higher-priority tasks. However, the system is designed to support priority implementation in the future. It is considered fulfilled with a few non essential features missing.
- **FR2 – Easily add new 'Event Nodes':** New nodes can be added efficiently using the JSON configuration system, allowing a new setup to be completed within minutes (see Code Snippets 6, and 7). Event nodes containing simple models may occasionally raise exceptions due to syntax errors when writing C# code directly into simple models, this can be further mitigated with additional development such as a DSL and or a GUI. This requirement is considered fulfilled, with room for implementation improvements. However, the system does not currently support adding event nodes at runtime. While this feature could offer value for example, by preserving the current state of active nodes, it was deprioritised in favour of more critical functionality.
- **FR3 - Quickly initialise REPS using a configuration file:** When excluding syntax errors, the system starts reliably using the configuration file. This functionality is considered stable and effective.
- **FR4 – Support for ML models in 'Event Nodes':** Successfully implemented with two ML models: SVM and RF. The current design allows each node to contain and manage its own model, making it straightforward to incorporate additional models in the future. The only potential limitation concerns models with high memory usage, such as KNN, which may impact performance.
- **FR5 – Event Detection Capability:** Successfully implemented. While the current implementation confirms that event detection is possible, further work is required to evaluate the accuracy and reliability of detection. The focus at this stage was on feasibility; assessing effectiveness will follow in future work.
- **FR6 – GUI:** Development was initiated but not completed. This requirement is not considered fulfilled.

- **FR7 – Automatic REPS creation:** Not initiated and therefore not fulfilled.

6.2 complexity

It was found that REPS exhibits low complexity. As complexity is an abstract concept in software, this report presents two arguments to support the claim:

- **Technological Complexity:** The REPS pattern requires minimal technological overhead. A communication platform similar to the MQTT standard and a data format such as JSON are sufficient to configure the system. Alternatively, a web interface or other GUI could replace manual configuration. As a result, the technical knowledge required to use the pattern is limited.

- **Configuration Overhead:** Defined as, knowledge a user need to attain to be able to fully configure REPS. The current implementation uses 30 unique keywords, keywords being predefined reserved identifiers which the REPS makes use of (Appendix 9.1). There is three non-essential keywords introduced for optimisation or to manage technical debt in the current implementation. One way to assess usability is to implement a basic function similar to a "Hello World". Since REPS is designed for decision-making rather than data generation, a simple addition example is provided (Code Snippet 6, page 22).

  For a more detailed example, a basic service has been developed to detect the mass creation of bot accounts on a website (Figure 10).

6.3 Measurements

This section discusses CPU and memory usage, presenting observations relevant to likely scenarios where REPS may be deployed. Latency is excluded, latency being the average time from event actually starts to REPS makes a determination. Latency depends largely on the update rate of REPS and network transport times. Program size is also not considered critical due to varying edge cases in different environments; for reference, the latest Debian Docker image occupies 231.81 MB.

6.3.1 CPU Usage

Each node uses very little processing power. Usage varies depending on the model, but the current implementations are so lightweight that a reliable assessment cannot be constructed. Nodes run in parallel, utilising all available CPU threads. When the number of nodes exceeds the number of CPU cores, REPS will use 100% of the CPU by processing multiple nodes concurrently, scheduling tasks as resources become available.

This can be problematic, as it may not be desirable to utilise all available resources unnecessarily. Therefore, some form of resource limitation should be considered in future work to prevent excessive hardware usage.

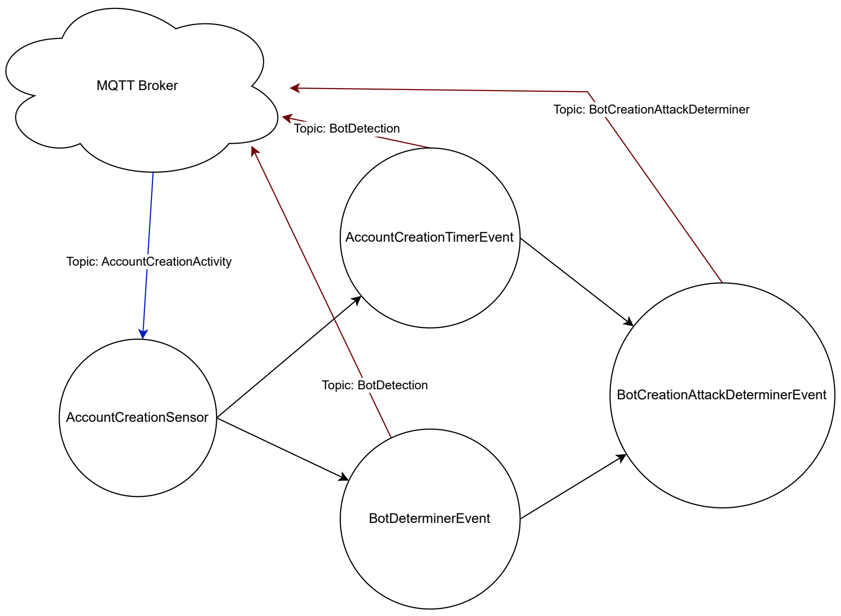
MQTT Broker
Topic: BotCreationAttackDeterminer
Topic: BotDetection
Topic: AccountCreationActivity
AccountCreationTimerEvent
Topic: BotDetection
AccountCreationSensor
BotCreationAttackDeterminerEvent
BotDeterminerEvent

Figure 10: node setup to determine whether an attacker is creating a large amount of bots

The current implementation writes extensively to the console to actively monitor the system's state. Approximately 52% of CPU usage is attributed to logging. While retaining some logging may prove useful for future debugging, it appears to consume a significant proportion of resources and may not be justified under normal operation.

CPU usage is minimal and therefore not a primary concern for further analysis.

6.3.2 Memory Usage

The simple bot detection service (Figure 10) consumes between 65 and 70 MB of memory, of which 25.85 MB accounts for overhead, including communication with RabbitMQ, logging, and related processes. Running the same service without the simple model (a run-time compiled method) reduces memory usage to between 41 MB and 44 MB. Adding an additional event node with a simple model to the original service increases memory consumption to approximately 75 MB, indicating an overhead of around 10 MB. The SVM model uses approximately 4 MB with five features. The memory usage of the timed model is negligible.

Memory usage is a important metric to design around if need to develop similar solutions based on these readings.

6.4 Research Question 1: Advantages and Disadvantages of REPS

Although the goal was to prove a concrete answer to this question, the exploratory nature of the project and lack of comparisons between existing solutions limits the extend this question can be answered. However, this does not make the question unanswerable. The original intent was to provide software engineers with sufficient insight to determine the suitability of REPS to their specific use-case.

To make such determinations, an engineer would require detailed information about the systems behaviour and performance. Sections 6.3 and 6.2 aim to provide this context. Based on this information, engineers may be able to assess the potential advantages and disadvantages using REPS.

Therefore, while the research question is not fully answered through direct comparisons, it still addressed indirectly with the projects original goals.

7 Discussion

7.1 Simulator

The Simulator proved effective for testing and generating data to train models on. There were a few problems from iteration to iteration where the site which the test was run on would change and therefore the Simulator would have to change as well.

7.2 Hindsight

In this section, a overview of the good tactics and the tactics which a developer should consider avoiding when making REPS or similar systems.

7.2.1 Good tactics

Setting up a proper development environment for this project is crucial, developing for docker in mind allowed for separation from the developers environment and test REPS more efficiently. The thread count for this program has lead to unforeseen problems and deploying on docker lowered the risk of my system having any affect on the test. Deploying on docker also allowed me greater deal of control of memory and version control.

For this project, it was planned more than the developer would usually do for a project, and yet tech-debt is a larger problem than expected. This is attributed to two things, the C# Roslyn library, as requirements changed throughout this project, the method to compile the user specified function needed many changes and test to verify the behaviour.

Making a JSON interpreter for this project has boosted development speed and most likely has overcome the initial time-cost which it cost to make it.

Using MQTT fits very well with REPS and has not caused any problems.

7.2.2 Tactics to avoid

A logging tool was used which were developed in previous projects to help debugging. This tools was not developed to avoid race conditions which just caused more exceptions than it logged.

7.2.3 Other findings

I discovered that compiling an event node function in run time requires more resources as well as taken a large time to implement. While using a ML model has more use cases and is often times depending on the model, cheaper in memory usage.

7.3 GUI

A GUI is useful for an end consumer, but working in this project it was not worth developing. The idea for developing it for this project even tough it already was a low priority, was to speed up the development process for the rest of the project. It could increase understanding if a visual was developed for the nodes in the REPS and allow for a more quick creation of new configuration files to test. It turned out after one iteration of work that this would require too much time to finish to a be at a useful state.

7.4 Reflection

This project changed a lot from the start of the project, while I generally think the project was a success, there is so much more that could have been done. I'm disappointed that I did not manage to make comparisons with other industrial technologies, nor that I did not make more scenarios to test which could be used in the REPS. At the start I was unsure whether I would model REPS configs based on real data sets or digital twins, using the digital twins was very good for the first experiment. It revealed a lot of the advantages for this project, and I did believe that the product wouldn't be worth developing until I started seeing the results of the first experiment.

I would wish that I planned more time for experimentation, while the implementation generally kept up with the plan in terms of developing the product, when I started to experiment there would be a lot of small changes that would be needed and was only really revealed once experimentations started. It was for example at this point I noticed the value of having a timer node, which is more of a support node than a event node.

Simple models took too long to implement, they are memory heavy, they are useful in some scenarios but the ML models were useful in more scenarios, and lighter on the memory. I now know that this should have been a 'could have' on the MoSCoW analysis and should properly first have started developing on after experiments and implementation of the MLs.

If the project were to be redone, the development of simple nodes would be postponed until the first round of experiments was completed. More time would be allocated to experimentation to better evaluate the usefulness of the system. At present, the project serves more to demonstrate feasibility than to assess the degree of utility. While some conclusions have been drawn, they are somewhat abstract and based primarily on experience rather than quantitative data.

7.5 Future Work

If more work is put into this project, these are the areas which the work will go into.

7.5.1 Experimentations

For future, there is a need to performed further experimentations on what extend REPS might be useful. This project assess the viability of REPS and some of the advantages and disadvantages which can be determined at the stage before thorough experimentations.

7.5.2 DSL

A DSL could bring values in terms of further improving the configuration files. I would lower the risk of a user making syntax errors, it could maybe further increase in speed, simplicity and or readability of the configuration files. While it is not necessary, it could further this project. The DSL would reduce pain-points of the current configuration system and would increase the speed of the user.

7.5.3 GUI

A GUI could be useful, but currently there is a lack sufficient evidence to make that determination. Further development and experimentation would be required to properly assess its value. This is in conflict with the DSL, in terms of setting everything up except for perhaps the simple models which contains the run-time code. One option is to combine the DSL with the GUI where the DSL would only be for the simple model when you have to specify a code block.

8 Conclusion

In this project, the software pattern REPS was conceived and developed, a software pattern designed to make determinations where different types of data are interrelated. By combining MQTT, JSON and parallel node interpretation, the REPS pattern was implemented.

Through developing REPS I identified effective strategies and pitfalls, offering insight that can benefit future developers working on similar systems or researchers which will take this further. REPS enables decision making while maintaining low complexity allowing users to define systems using a small set of configuration keywords.

This project demonstrates that REPS is a viable approach for building system that make determinations. The next step is to evaluate REPS' usefulness in comparison with existing solutions. If successful, REPS could offer a developer friendly alternative.

9 Appendix

Github Repository: https://github.com/SSpedsbjerg/Master-Thesis/tree/Development

9.1 Number of keywords

This is a list of all of the keywords which REPS uses

1. BrokerInfo : An object containing required info for the broker
   (a) NotificationBroker : Where REPS should notify the user on an event
   (b) NotificationTopic : What topic it should notify
   (c) ReportBroker : Where it should report to
2. SensorNodes : An object containing all sensor nodes
   (a) ID : used to refer back to it
   (b) Name : Used to make error handling easier
   (c) Host
   (d) RoutingKey
   (e) Topic : What topic to report to
3. EventNodes : An object containing all event nodes
   (a) ID
   (b) Name
   (c) SensorNodes : An array of id for all sensor nodes which this node should retrieve data from
   (d) EventNodes : An array of id for all event nodes which this node should retrieve data from
   (e) ReportTopic
   (f) SupportedType : Determination of what type the node should work as, this keyword can in theory be removed but exists still due to tech debt.
   (g) Host
   (h) RoutingKey
   (i) Model : An object containing all information for the model
      i. ID
      ii. Type : What model type it is, this follows with their own keywords
      iii. isUsername : if you handle usernames or other similar text based input, you can use this. This is not needed for the pattern or a user, but I found it useful for optimisation purposes.
      iv. DebugMode : puts the model into debug mode
         A. simpel : A model which the user can define themself with code

- Function : A function in C# which will be compiled in runtime so the user can define themself.
- TriggerFunction : A logical function to determine when a event is triggered and should

B. timed : A model to find a average amount of unique
   - TimedTrigger : A value limit to determine whether a event has happend or not

C. Adaptiv : A model that can change its trigger limit depending on its usecase
   - QuantileCutoff
   - UpdatePercentage
   - It also shares it the same keywords as simpel

D. svm : Support Vector Machine model
   - TrainingData : Path to the trainingsdata

E. forest : Random Forest Modell
   - TrainingData
   - NumberOfTrees

References

[1] Nuget. Nugest packages sorted by most downloaded. 2025. URL: https://www.nuget.org/packages?q=&includeComputedFrameworks=true&prerel=true&sortby=totalDownloads-desc.

[2] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ““why should I trust you?””. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Disco (2016), 1135–1144. DOI: 10.1145/2939672.2939778.

[3] Zonghan Wu et al. “Connecting the dots: Multivariate time series forecasting with Graph Neural Networks”. In: Proceedings of the 26th ACM SIGKDD International Conference (2020). DOI: 10.1145/3394486.3403118.