Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, P. Lendermann, eds.

JEOPARDY ASSESSMENT FOR DYNAMIC CONFIGURATION OF COLLABORATIVE MICROSERVICE ARCHITECTURES

Glen Pearce

Defence Science and Technology Group
Third Avenue
Edinburgh, SA 5111, AUSTRALIA

Alexis Pflaum
Dumitru Alin Balasoiu
Claudia Szabo

The University of Adelaide
North Terrace
Adelaide, SA 5000, AUSTRALIA

ABSTRACT

Microservice architectures, which are lightweight, flexible, and adapt easily to changes, have recently been considered for system development in military operations in contested and dynamic environments. However, in a military setting, the dynamic configuration of collaborative microservices execution becomes critical, and testing that microservice configurations behave as expected becomes paramount. In this paper, we propose a complex jeopardy metric and reconfiguration process that dynamically configures collaborative algorithms running on multiple nodes. Our metric and proposed scenarios will allow for the automated evaluation of microservice configurations and their re-configuration to suit operational needs. We evaluate our proposed scenario, metric, and various reconfiguration algorithms to show the benefits of this approach.

1 INTRODUCTION

It is very often that crisis, rescue and military operations take place in contested and dynamic environments, where the computational resources are limited in either computational, power or network capacity, the network is under attack, or the operational context changes rapidly. Military operations in particular have strict service guarantees within these environments, need to scale capably, and continue operating at all costs. In future battlespaces, the intelligent systems on-board vehicles will also require preparation to avoid conflict in action. The system complexity will make manual configuration for optimal performance impossible, in particular when conflicting operational needs are exerted. In addition, the workload to monitor and manage these systems can also impact crew performance and increase stress. As such, there is a need for flexible and scalable architectures, that can be configured easily within operational and mission contexts.

Microservice architectures are an appealing solution to address the architectural challenges described above. Microservices are decoupled software with limited context, with well-defined and explicitly published interfaces (Newman 2015; Dragoni et al. 2017). Each microservice is fully autonomous and full stack, thus leading to loose coupling, cohesion, and flexibility. This leads to faster delivery, improved scalability and greater autonomy (Newman 2015; Dragoni et al. 2017; Dragoni et al. 2017). Fast delivery is achieved as microservices are typically packaged and deployed in the cloud using lightweight containers supported by a full software stack. In addition, microservices benefit from automated software integration and delivery. Automated software integration and delivery also allows for greater autonomy, and the full deployment within a cloud setting exploits existing solutions for scalability and elasticity (Chen et al. 2007; Mendonça et al. 2018; Mendonça et al. 2019).

Within a contested and dynamic environment, a deployed microservice architecture should be autonomic and self-configuring, while at the same time not inhibiting mission and operational goals. In traditional

microservice deployments, microservice interaction might carry computational cost such as that incurred by sending messages over the network. However the concept of risk, such as detection or attack risk, has yet to be considered. In addition, the evaluation of microservice architectures and service architectures more broadly only consider generic infrastructure metrics, without considering mission effectiveness or application specific metrics (Taherizadeh et al. 2018; Szabo et al. 2020).

In this paper, we propose the use of a mission specific jeopardy metric, which combines the risk and utility of specific microservice interactions within a sample mission Plan that generates scenarios used for simulation. Our contributions are threefold:

- A simulation scenario generation tool that generates complex scenarios from a simple set of Plans
- A jeopardy metric and assessment algorithm that combines the risk and utility of specific microservice interactions within a mission Plan.
- The prototype implementation of a configuration optimization algorithm and a prediction algorithm, and their extensive experimental evaluation in simulation, showing the ease of their integration within our tools.

2 RELATED WORK

Our work is closely related to service orchestration and coordination, in that it proposes context specific measures to drive service orchestration and service deployment configuration. Orchestration is critical for the effective design and deployment of microservices however few research works focus specifically on orchestration of microservices, relying instead on existing service oriented architecture and business process modeling approaches. We discuss in the following several relevant works within this space.

To address the challenges posed by the dynamic nature of IoT devices and the variations in workload, Taherizadeh et al. (2018) propose a capillary computing architecture, which relies on a capillary container orchestrator and an Edge/Fog/Cloud monitoring system to make decisions. An Edge node is an IoT node that has some computing capabilities; Fog nodes are cloud computing infrastructures in the vicinity of Edge nodes. The proposed approach follows a MAPE-K (Monitor-Analyze-Plan-Execute over shared Knowledge) loop (Kephart and Chess 2003; Brun et al. 2009) where Edge computing resources can run all the necessary microservices. When an IoT device moves, the running container will be terminated, and another instance will be spun on another Edge node in proximity to the IoT device. The approach relies on a set of metrics related to infrastructure, network and application performance such as CPU, memory, disk, delay, packet loss, and jitter among others. The approach is validated using a smart application used for car automation, and the employed metrics include response time and runtime. However the case study has a small number of microservices and as such the agility of the orchestrator cannot be properly evaluated.

Medley (Yahia et al. 2016) is an Event-driven platform for service composition based on a domain-specific language (DSL) for describing orchestration. Medley relies on a service specification that includes endpoints, operations, and data type, for all services that will be used in the orchestration. Before defining a composition, Medley requires developers to register the information (e.g. endpoints, operations, data types) about each microservice that will be used during the orchestration. The approach is evaluated using a simple simulated example. Similar to Medley, Microflows (Oberhauser 2016) requires extensive information about microservices. Proposed as an alternative to rigid Business Process Models, Microflows propose to use Belief-Desire-Intention agents and graph-based representations to describe goals and constraints on microservice compositions. Both approaches are promising, however they require complete specifications related to the microservice at design time. Due to the dynamic nature of a microservice, implying that it can be deployed, replicated or reallocated during the application execution, it is not possible to determine the microservice endpoint at design time.

Beethoven (Monteiro et al. 2018) aims to address these challenges by using service discovery patterns to dynamically determine the location of microservices. The authors also proposed Partitur, an orchestration DSL that represents the composition as a workflow with tasks and Event handlers. The approach is evaluated

using a sample Customer Relationship Management system for an investment bank, comprised of seven microservices.

Alam et al. (2018) propose a multi-layer orchestration architecture for IoT microservices. The architecture is comprised of three layers, namely, smart IoT devices (Edge), Gateways (Fog) and Enterprise systems. The smart IoT devices represent the sensing layer and contain microservices that are deployed in lightweight Docker containers. The gateway layer performs the monitoring of the sensing layer, and the enterprise systems layer performs long-term analytics and orchestration. The approach is evaluated using a typical smart home appliance where two different end-devices are periodically communicating with central servers deployed in the cloud. The end-devices work as cyber-physical systems, that can be remotely controlled and can also interact directly with the user.

OpenTosca (Binz et al. 2012) is a well-known open source ecosystem for the Topology and Orchestration Specification for Cloud Applications (TOSCA). OpenTosca is divided into three parts: a TOSCA runtime environment (OpenTosca container), a graphical modelling TOSCA tool (Winery) and a self-service portal for the application available in the container (Vinothek). Although OpenTosca is a generic framework, it does not support runtime orchestration, which we see as a precursor to adaptive microservices architectures. Cloudify builds upon a TOSCA and provides access to multiple clouds and a complete framework to describe microservices and execute them either in Docker containers or on cloud metal, with minimal monitoring support.

3 COLLABORATIVE SERVICES FOR THREAT REDUCTION

Current vehicle systems include software services engineered to provide a range of capabilities tailored to the hardware systems deployed with the vehicle. Systems are configured by SMEs (Subject Matter Experts) to provide maximum protection and coverage. Vehicle and subsystem localisation can be improved through collaborative data fusion. By collaborating with heterogeneous systems, they can take advantage of subsystem redundancy and increase team resilience when individual systems fail or are under attack (Henderson et al. 2020). However, service collaboration and coordination to improve one operational aspect might affect the performance of other services. An example is signature management. EM radiation signature can be minimised with the trade off in communication range and/or data throughput (Shaw et al. 2020).

Within this work we have selected six main services, namely, *Communication*, *Radio Frequency Scanning*, *Communication Jamming*, *GPS Spoof Detection*, *Shot Detection*, and *Radar*, because they are widely used within various architectures and their combined use leads to various trade-offs.

- *Communication*: Using mesh networking and beam forming the range of communication of the convoy can be extended. More collaborators can increase the utility of this service, but only if the topology allows beneficial collaboration. Furthermore, electromagnetic emissions can expose the convoy to the threat of detection.
- *Radio Frequency Scanning*: Electromagnetic emissions can be detected by anyone. Enemy communications can be detected by specialist equipment. The utility of this microservice should increase with the number of collaborators, as more frequencies can be monitored.
- *Communication jamming*: The enemy relies on radio messages to coordinate attacks and trigger pre-laid munitions. Collaborative jamming will improve the chance that these may be interrupted. However, it will also interrupt efforts to scan for communications.
- *GPS Spoof detection*: Spoofing can be detected through a number of means. It's performance is improved through collaboration, and distributed nodes open the opportunity to triangulate the malicious signal.
- *Shot detection*: Audio triangulation of audio shock-waves can categorise and localise shot trajectories. This can alert team members of pending ambush, or a failed infiltration.

- *Radar*: Coordination of arcs can increase the area of coverage and data fusion techniques can be used to triangulate targets.

For each of these services the shared utility is increased when more collaborators are contributing. However, resources such as spectrum availability, power and compute capacity are limited. Furthermore, some capabilities may interfere with others. Worse still, the capabilities on one vehicle may interfere with those on another. It is our objective to provide autonomic control for these services to improve utility for the situational context.

3.1 Service Interaction and Utility Calculation

To model these properties we have created an equation for independent utility (1) and cross reference map of microservice interactions, shown in Table 1.

Table 1: Service Interaction Matrix, where 0 denotes no interaction, +1 indicates a positive interaction, and -1 indicates a negative interaction.

	C	**S**	**J**	**G**	**D**	**R**
(C)ommunications	0	0	-1	+1	0	+1
(S)canning	0	0	-1	+1	0	-1
(J)amming	-1	-1	0	-1	0	0
(G)psSpoofDetect	+1	+1	-1	0	0	0
Shot(D)etect	0	0	0	0	0	0
(R)adar	+1	-1	0	0	0	0

This table is used to generate a single value for utility. First, the independent utility U_I for each service is specified by an increasing curve dependent on the number of collaborators n for that service s. The values are defined by SMEs but may be approximated by a sigmoid curve such as the logistic function. This value is then multiplied by the relative service value S_R to give the independent utility across all nodes for that service as shown in Equation 1.

$$U_I(s) = S_R \sigma_s(n) \tag{1}$$

The positive and negative interactions between running services $I(s,\vec{s})$ are captured in Table 1, where: 0 means no Change; $+1$ means *positive interaction*: increase the larger value by half the lower value; and -1 means *negative interaction*: decrease the larger value by half the lower value. The total of the independent utility values and all pairwise interaction values give a single utility value U. For a set of microservices $S = \{s\}$ this can be given by:

$$U = \sum_{s \in S} U_I(s) + \sum_{\vec{s}, s \in S} I(s,\vec{s}) \tag{2}$$

Equations 1 and 2 generate a utility metric that does not simply reward more services being run but also captures subject matter expertise as to when services affect each other.

3.2 Operational Context and Autonomic Control

Autonomic system control enables vehicle systems to self-configure as needed to achieve mission objectives or desired performance metrics. The key difference between these and autonomous systems is that they do not direct the system's actions, but configure the on-board systems to best address the operational context.

We define this operational context as a set of environmental states that change throughout the mission. Events are triggered by notional sensors, that is, they can be derived from equipment or inferred from current knowledge. Within the scenario, the convoy will move through a series of Events as defined below.

- On Road: The convoy must move through an area with limited freedom of formation. This will reduce the utility of collaborative services that rely on convoy formation to provide utility and increases the chance of being detected or ambushed.
- In Range: The convoy is within the expected area of operation of enemy forces. They can expect a greater chance of interacting with the opposing force.
- Detect GPS Spoof: This is an attempt to disorient the blue force and reduce their confidence in their location. It is a clear indication to the convoy that the enemy is Planning an ambush.
- Detect Shot: The convoy is clearly within an enemy area of operation. Through collaboration between nodes, the convoy can determine whether it is directly under threat.
- Engagement: The convoy is actively engaging with the enemy. The is a 100% chance that the convoy has been detected.

The combination of environment states describe the context against which the autonomic control must optimize its configuration. However, within the simulation we do not have control over the Event that will occur next.

The occurrence of a sensor activation and implicitly an Event sometimes might imply that other sensor activations are imminent and might also lead to increased risk. Table 2 captures the relationships between the observed Events and this implied risk as derived from conversations with SMEs (empty cells are a zero value). Examples include the effect of the convoy being detected, the effect of the convoy being fired at, the likelihood of the convoy being detected, and lastly the likelihood of the convoy being attacked.

Table 2: Implied Risk Values. Sensor activation by row with the implication of risk in the columns.

	InRange	OnRoad	GPS Spoof Detected	Shot Detected	Engagement
InRange	**8**		2		2
OnRoad	1	**12**	2	3	4
GPS Spoof Detected	1		**16**		
Shot Detected	2			**20**	4
Engagement	2			5	**32**

The risk attributed to sensor activation is not a linear summation as the risk implications are not additive or orthogonal. Each sensor indicates a confirmation of enemy Activity and, given knowledge of likely operating procedures, an implied increase in likelihood of other sensors being triggered. Multiple sensors can be triggered which confirm these implications, but should not multiply the risk value. Using Table 2, we can calculate a single Risk value R from multiple sensor Events e (the rows from Table 2) using Equation 3.

$$R = \sum_{e} max(\hat{I}(e)) \tag{3}$$

where $\hat{I}$ gives the implied risk for each Event.

3.3 Jeopardy for Risk Assessment

As discussed above, there are a number of services that could be active on various vehicles or nodes at any point in time. Each of the services has several specific requirements and capabilities, and the collaborative execution of these services should lead to improved mission outcome. However, the execution of each individual microservice carries a certain amount of risk, as each microservice execution potentially increases the probability of enemy discovery. This could be achieved for example through tracking emissions, digital footprint, or infrared scans.

There are many factors that contribute to the calculation of risk and utility, including the current Events in the operational context, environmental sensors as well as the specific microservice deployment on each node. We introduce the concept of Jeopardy to capture the carried risk of selecting a future microservice configuration within an autonomic configuration scenario. Jeopardy is used to calculate the risk and utility of a future specific configuration, and is used in the automated selection of that configuration. Whether the jeopardy of a specific configuration and inherently the configuration is accepted depends on the mission Plan, which is not the focus of this current work.

Jeopardy as a numeric measure captures the carried risk in selecting a configuration for a given situation. The node's sensors determine the situation they are experiencing, by taking readings of the environment. Based on the situation, nodes can turn on or off services to try mitigate some of the risk. Services are not always beneficial to the nodes, for example the use of radar needs to balance learning about the environment against giving away their position.

We define a jeopardy metric that combines risk from Equation 3 and utility from Equation 2 as follows:

$$J = \frac{R}{U} * 100, U > 0 \tag{4}$$

4 SCENARIOS FOR MICROSERVICE CONFIGURATION ASSESSMENT

We developed the following exemplar for the evaluation of microservice configuration using our proposed jeopardy metric. In our sample scenario, Blue forces have plotted a course that skirts known enemy defensive lines. They will observe the enemy from cover and wait until security patrols change shifts. Blue forces will then breach the defensive structures and secure the facility. This simplified set of Plan States was developed from (Curtis and Hobbs 1997). While doctrine separates Activities such that each term uniquely describes the context in which it occurs, we have extracted the underlying relevant Activities that can be conducted simultaneously as defined below:

- *Manoeuvre*: The convoy moves through an area with a communicated set of intentions and contingencies. The commander will use their knowledge to select appropriate formations and when to refine orders.
- *Observe*: Gathering information about the enemy, usually while avoiding discovery.
- *Breach*: Breaching is the employment of a combination of tactics and techniques to advance an attacking force to the far side of an obstacle.
- *Strike*: Strike describes any combat action. As it is a key focus of tactical studies it is usual defined in more detail with terms that entail context.

By constructing mission Plans from such high level states, we can describe common tactics using a combination of Plan states. For instance, a Patrol is a combination of Manoeuvre and Observe. This layering of states allows us to use to individually map probabilities for changes in environmental states.

4.1 Scenario Generation from Plans

Forces have a concept of what they will be experiencing when they are sent out on a mission, even if they do not know the mission specifics. For example, when travelling towards an adversary's known location, it is known that for the first part of the trip the likelihood of contact is low. This aspect can be easily modelled.

We use the Activities above and consider the probability of Events occurring during the Activity period that can be defined. For example, the Event of 'shot detected' is more likely to occur during a breach or strike Activity than during an observe Activity. A mission Plan is a perception of the future and as such it is likely to be inaccurate, due to several unforeseen Events. For example, when manoeuvring to observe an adversary, the times of arrival might be affected by the terrain. To account for this, we introduce randomness to the start/end time of Activities, which is sampled from normal distributions. We set the mean

of the distribution over the perceived start of stop time of an Activity, then change the size of the standard deviation. This is shown in Figure 1, with the time series on the x-axis and probability of membership in a given activity on each y-axis.

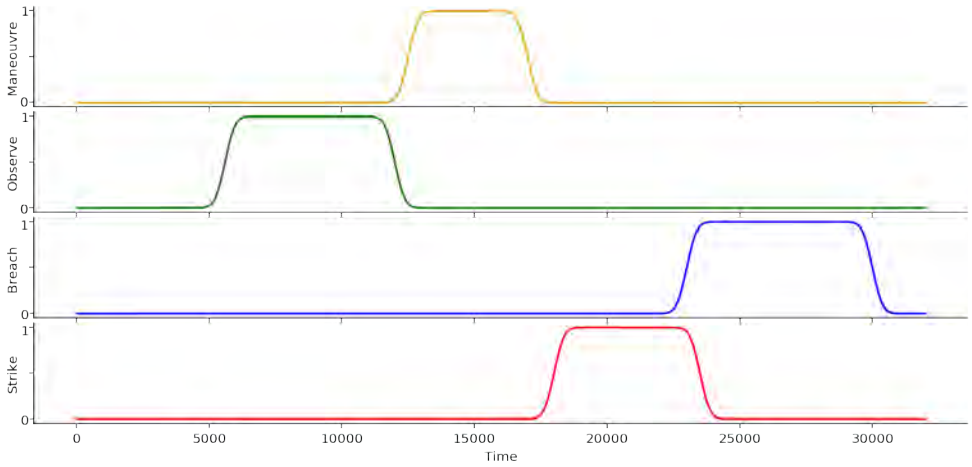

Figure 1: A Plan showing Activities starting and ending with a normal probability distribution.

4.1.1 Use of Plans and their Implementation

We employ the Plans in creating scenarios that can be then used with different decision algorithms that use jeopardy as a metric of their effectiveness. Examples of two such algorithms are given in Section 5. Simulation scenarios are created by a combination of a Plan when Activities are thought to happen and an estimation of confidence of the Activity and Events as shown in Table 3.

The applied confidence to a Plan contains all of the probabilities associated with Activities and Events. For each Activity, the standard deviation is specified for the starting and stopping times, and probabilities of Events starting and stopping are entered. The confidence contains the standard deviation for the normal distribution of when Activities will proceed and the probability of an Event starting or stopping given a certain Activity happening. Given more than one Activity can happen at the same time, the higher of the Events probabilities are used. Once an Event has changed state, we impose a limit on the simulation of when the Event can change again in the form of a cool down period.

5 EXPERIMENTAL ANALYSIS

In this section, we employ two optimization algorithms on different parts of our pipeline. We minimize jeopardy and identify the best service configuration using an evolutionary algorithm. We employ a neural network to predict the next Event within the scenario, allowing the system enough time to create its next configuration. Our experiments use a four node scenario, on which six services that can collaborate are deployed. As defined above, these services are Communication (C), Scanning (S), Jamming (J), GPS Spoof Detected (G), Shot Detected (D), and Radar (R).

Each microservice runs on a node within the system and is modelled as a Python Flask microservice. The microservice exports its metrics into Prometheus as real-time time-series data via a HTTP GET method. The microservices are each deployed in a Kubernetes pod. In this prototype implementation, the services themselves are simple computations with stateless and contextless interactions, however in future work this will be expanded to include mission specific interactions. Each physical node is modelled as a Kubernetes node using Kubernetes' node affinity feature and we assign microservices to pods according to

Table 3: Activity and Event associated probabilities and standard deviations.

		Manoeuvre	Observe	Breach	Strike
Standard Deviation (min)		12.0	6.0	12.0	6.0
Situation	Event	On to Off Cooldown Time	Off to On Cooldown Time	On to Off Probability	Off to On Probability
Manoeuvre	OnRoad	15	15	0.95	0.05
Observe	OnRoad	15	15	0.95	0.1
Breach	OnRoad	15	15	0.95	0.02
Strike	OnRoad	15	15	0.9	0.1
Manoeuvre	InRange	15	15	0.25	0.05
Observe	InRange	15	15	0.15	0.1
Breach	InRange	15	15	0.05	0.02
Strike	InRange	15	15	0.2	0.1
Manoeuvre	ShotDetect	15	15	0.25	0.05
Observe	ShotDetect	15	15	0.15	0.1
Breach	ShotDetect	15	15	0.05	0.02
Strike	ShotDetect	15	15	0.2	0.1
Manoeuvre	SpoofDetect	15	15	0.25	0.05
Observe	SpoofDetect	15	15	0.15	0.1
Breach	SpoofDetect	15	15	0.05	0.02
Strike	SpoofDetect	15	15	0.2	0.1
Manoeuvre	Engagement	15	15	0.25	0.05
Observe	Engagement	15	15	0.15	0.1
Breach	Engagement	15	15	0.05	0.02
Strike	Engagement	15	15	0.2	0.1

the microservices status. Turning on a microservice on a node is modelled by creating a Kubernetes pod and assigning it to the node, while a turning it off is modelled by terminating the microservice container.

5.1 Genetic Algorithm

We employ a genetic algorithm to optimize the microservice configuration (on/off) that minimizes the jeopardy metric defined in Equation 4. The reconfiguration algorithm is triggered during the scenario vignette whenever a new Event occurs.

Inspired by the natural selection that occurs in nature, the generic algorithm starts with an initial population of individuals. An individual genotype is represented as a 24 digit binary string where the gene value stores the status of the corresponding microservice (0 - off, 1 - on). In this four node scenario, every 6 bits in the individual chromosome represents the microservice configuration of a node, respectively in this order (C)ommunication-(S)canning-(J)amming-(G)PS Spoof detected-Shot (D)etected-(R)adar.

The algorithm starts with an initial set of 100 randomly generated candidate solutions as the initial generation. In every generation, the fittest individuals are selected (with the lowest jeopardy) and they evolve by creating offsprings using nature-inspired operations, namely, *mutation* and *crossover*. Similar to how in nature we observe that a gene can change, thus maintaining diversity, the algorithm randomly *mutates* or changes a gene value with a low probability. Also inspired from biology, we use the *crossover* operator, also known as recombination, where offsprings are created by mixing genes from both parents. Using a stochastically chosen point called crossover point, the offsprings are created by exchanging each gene until the crossover point. The crossover point is selected using Python's inbuilt pseudo-random generator `random.randint()` function as a value between 2 and 23.

At each generation, five random immigrants were inserted in the population to stimulate diversity. We used a mutation probability of 20 percent, a crossover probability of 50 percent and selection. The process

continues for up to 20 generations or until up to 6 consecutive generations that have no improvement. At the end of the run, the best fit individual is selected and the microservices are reconfigured accordingly.

5.2 Neural Network

We explore in these experiments whether a neural network can be within our scenario. In these experiments, we focus on the cost of changing between microservice configurations, which requires service shutdown, pod configurations, and microservice migrations. To alleviate this cost, we train a neural network to predict the likelihood of future Events. Knowing what future Events are likely to occur allows the system a lead-time to switch between active microservices. We employ a neural network in which we feed example scenarios created from a Plan, enabling the network to predict likelihoods of Events happening in the near future.

We employ the DeepLearning4J framework to create a shallow neural network containing two internal layers, namely, a Long Short Term Memory (LSTM) layer and a Dense Layer. The LSTM layer could be effective in retaining the current state of the system as Activities span over a period of time, and Events changing have varied cool down times. The LSTM layer could be good in retaining the current system state, as the Activities span over a period of time.

The system was trained with up to 90% of a hundred thousand simulations created from the same Plan with corresponding confidences. Leaving 10% for validation/testing which shared the same data set. A simulation file contains the data of the current Events, the expected Plan of Activities, and the actual Activities being experienced per second of the simulation. The testing simulations were created in the same manner, but were not used in the training. Given the Events and proposed Plan we hoped the system would be able to determine when specific Activities occurred. The creation of the data for the system was achieved by modifying the existing simulation files from containing current Events to be reacted to also include the estimate Plan and the Events experienced by the system 5 minutes into the future.

5.3 Results

5.3.1 Optimizing Microservice Configurations

We ran 10,000 simulation scenarios and for each triggered Event during the course of the scenario a genetic algorithm was launched to find the best fit individual, i.e. the microservice configuration with smallest jeopardy. An example evolution of the best jeopardy found throughout generations of our genetic algorithm for a generated simulation scenario can be seen in Figure 2.

This uses the generated simulation scenario `Plan 91` with the list of Events shown in Figure 3. The first Event at 0:21:22 simulation time is *ShotDetect On* and this triggers the genetic algorithm run that found the best configuration with jeopardy 107, specifically with the following microservices turned on in n1: *GPS Spoof Detected* ; node n2:*GPS Spoof Detected — Shot Detected — Radar* ; node n3: *Communication — Jamming — GPS Spoof Detected*; node n4: nothing active. The simulation ends after the last Event, *Engagement Off*, at simulation time 1:51:25 where the last instance of the genetic algorithm is used for find a configuration with jeopardy value of 373, specifically with the following microservices turned on in node n1: *Scanning — Shot Detected — Radar*; node n2: *Communication — Scanning — Shot Detected*; node n3: *Shot Detected* ; node n4: *Communication* after 8 generations.

5.3.2 Predicting Future Events

Throughout our experiments the input, LSTM and dense layers had the same number of inputs and output, 13, the number of inputs from each record of the simulation. The output layer consisted of a single number between 0 to 2^n where n is the number of possible Events. The activation functions used was tanh(5) and softmax(6).

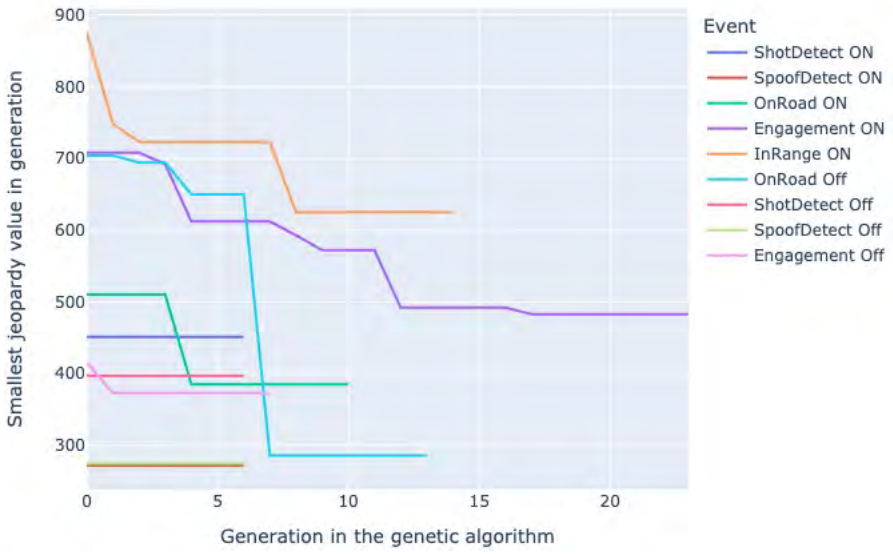

Figure 2: Optimizing Jeopardy as Various Events are Activated in Plan 91.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5}$$

$$f_i(x) = \frac{e^{x_i - shift}}{\sum_j e^{x_j - shift}} \tag{6}$$

where $shift = max_i(x_i)$.

The system would consistently converge to predicting all Events would be occurring 5 minutes into the future. With this specific choice the algorithm's accuracy was 24%, a recall of 3% and an F1=38%.

We believe the shallow neural network did not have the ability to encapsulate the number of changing dependant variables to make a more nuanced prediction. The changing variables included the hidden actual Activities being experienced which the estimated Activities were to be estimating.

5.4 Discussion

When optimizing microservice configurations using the genetic algorithm, we ran each scenario 10 times. For each Event activation in the scenario, the genetic algorithm ran for up to 20 generations or until 6 consecutive generations had no better score. As the generic algorithm tries to optimize on a single objective, we discovered that in some cases the lowest jeopardy might not be the only objective to optimize, as the lowest found jeopardy for a current Event could result in the jeopardy for the next Event in the scenario being higher. This also makes the optimization reactive; something the neural network approach hoped to resolve.

Our machine learning algorithm showed poor results but it also showed promise, both in its demonstrative application but also as with deeper neural networks and a better encoding of Activities better predictions can be obtained. Another issue is that the scenario generation methodology produces discrete Events which are difficult for gradient descent approaches to learn. Jeopardy as a metric is still appealing because it captures mission context aspects in a generic manner, lending itself well to other learning approaches, such as reinforcement learning. The study of the optimization of service configurations using machine learning approaches is part of our future work.

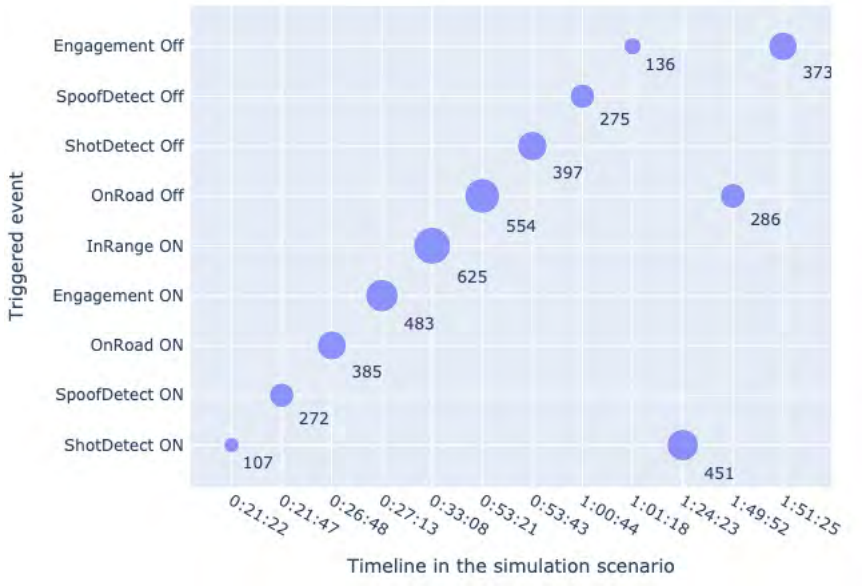

Figure 3: Event Sequence in Plan 91.

6 CONCLUSION

We propose in this paper an approach to evaluate the mission effectiveness of autonomic microservice configurations. Towards this, we employ a scenario generation approach from mission states. We define a jeopardy metric that considers both the risk and utility of the collaboration and concurrent execution of multiple services. We showcase the versatility of the scenarios and proposed jeopardy metric using a prediction and optimization algorithm, namely, a neural network and an genetic algorithm. The neural network predicts, based on the jeopardy value, which Event will be activated next in the scenario. The genetic algorithm optimizes the microservice configuration to minimize the jeopardy value.

While our approach to incorporate mission effectiveness metrics within autonomic microservice configurations is appealing, several avenues for future work exist. The current implementation and metric does not consider the need for microservices to coordinate, which results in constraints that specific services are running at the same time. In addition, microservice communication is not explicitly considered within the jeopardy calculation and sensor activation Events are currently only considered in pairs.

REFERENCES

Alam, M., J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen. 2018. "Orchestration of Microservices for IoT Using Docker and Edge Computing". *IEEE Communications Magazine* 56(9):118–123.

Binz, T., G. Breiter, F. Leyman, and T. Spatzier. 2012. "Portable Cloud Services Using TOSCA". *IEEE Internet Computing* 16(3):80–85.

Brun, Y., G. D. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. 2009. "Engineering Self-Adaptive Systems Through Feedback Loops". In *Software Engineering for Self-Adaptive Systems*, 48–70. Berlin, Germany: Springer.

Chen, C., S. B. Nagl, and C. D. Clack. 2007. "Specifying, Detecting and Analysing Emergent Behaviours in Multi-Level Agent-Based Simulations". In *Proceedings of the Summer Computer Simulation Conference*, 969–976. New York, USA: Association for Computing Machinery, Inc.

Curtis, N. J., and W. S. Hobbs. 1997. "Characterisation of Infantry Section and Platoon Activities". Technical report, Defence Science and Technology.

Dragoni, N., S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. 2017. "Microservices: Yesterday, Today, and Tomorrow". In *Present and Ulterior Software Engineering*, 195–216. Berlin, Germany: Springer.

Dragoni, N., I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina. 2017. "Microservices: How to Make Your Application Scale". In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, 95–104. Springer.

Henderson, J., J. Trumpf, and M. Zamani. 2020. "A Minimum Energy Filter for Distributed Multirobot Localisation". *IFAC-PapersOnLine* 53(2):4916–4922.

Kephart, J. O., and D. M. Chess. 2003. "The Vision of Autonomic Computing". *Computer* 36(1):41–50.

Mendonça, N. C., D. Garlan, B. Schmerl, and J. Cámara. 2018. "Generality vs. reusability in architecture-based self-adaptation: the case for self-adaptive microservices". In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, 1–6. New York, USA: ACM.

Mendonça, N. C., P. Jamshidi, D. Garlan, and C. Pahl. 2019. "Developing self-adaptive microservice systems: Challenges and directions". *IEEE Software* 38(2):70–79.

Monteiro, D., R. Gadelha, P. H. M. Maia, L. S. Rocha, and N. C. Mendonça. 2018. "Beethoven: An Event-Driven Lightweight Platform for Microservice Orchestration". In *Proceedings of the European Conference on Software Architecture*, 191–199. Berlin, Germany: Springer.

Newman, S. 2015. *Building Microservices: Designing Fine-Grained Systems*. " O'Reilly Media, Inc.".

Oberhauser, R. 2016. "Microflows: Lightweight Automated Planning and Enactment of Workflows Comprising Semantically-Annotated Microservices". In *Proceedings of the Sixth International Symposium on Business Modeling and Software Design*, 134–143. Setubal, Portugal: SciTePress.

Shaw, P., B. Campbell, and B. Sims. 2020. "Enhanced Distributed Connectivity Control in Low Probability of Detection Operations". In *Proceedings of the Military Communications and Information Systems Conference*, 1–6. Piscataway, New Jersey: IEEE.

Szabo, C., B. Sims, T. Mcatee, R. Lodge, and R. Hunjet. 2020. "Self-Adaptive Software Systems in Contested and Resource-Constrained Environments: Overview and Challenges". *IEEE Access* 9:10711–10728.

Taherizadeh, S., V. Stankovski, and M. Grobelnik. 2018. "A Capillary Computing Architecture for Dynamic Internet of Things: Orchestration of Microservices From Edge Devices to Fog and Cloud Providers". *Sensors* 18(9):2938.

Yahia, E. B. H., L. Réveillere, Y.-D. Bromberg, R. Chevalier, and A. Cadot. 2016. "Medley: An Event-Driven Lightweight Platform for Service Composition". In *Proceedings of the International Conference on Web Engineering*, 3–20. Berlin, Germany: Springer.

AUTHOR BIOGRAPHIES

GLEN PEARCE is a Research Engineer at the Defence Science and Technology Group, Australia. His work focusses on the use of self-organisation and collaboration in distributed systems. His email address is glen.pearce@defence.gov.au.

ALEXIS PFLAUM is a PhD Candidate in Computer Science within the School of Computer Science at The University of Adelaide in South Australia. He is a senior software engineer with over 10 years of industry experience ranging from anti-spam software, business process optimization to mobile apps developmen. His email address is alexis.pflaum@adelaide.edu.au.

DUMITRU ALIN BALASOIU is a software engineer working in the Complex Systems Research Group at the University of Adelaide. His interests include data science, security and microservices. His email address is dumitru.balasoiu@adelaide.edu.au

CLAUDIA SZABO is an Associate Professor in the School of Computer Science and the Lead of the Complex Systems Research Group at the University of Adelaide. Her research interests lie in the area of complex systems and on using simulation to identify and validate their emergent properties. Her email address is claudia.szabo@adelaide.edu.au.