Automated Management of Collections of Autonomic Systems

Thomas J. Glazier, David Garlan, Bradley Schmerl
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
{tglazier, garlan, schmerl}@cs.cmu.edu

***Abstract*—Many applications have taken advantage of cloud provided autonomic capabilities, commonly auto-scaling, to harness easily available compute capacity to maintain performance against defined quality objectives. This has caused the management complexity of enterprise applications to increase. It is now common for an application to be a collection of autonomic sub-systems. Combining individual autonomic systems to create an application can lead to behaviors that negatively impact the global aggregate utility of the application and in some cases can be conflicting and self-destructive. Commonly, human administrators address these behaviors as part of a design time analysis of the situation or a run time mitigation of the undesired effects. However, the task of controlling and mitigating undesirable behaviors is complex and error prone. To handle the complexity of managing a collection of autonomic systems we have previously proposed an automated approach to the creation of a higher level autonomic management system, referred to as a *Meta-Manager*. In this paper, we improve upon prior work with a more streamlined and understandable formal representation of the approach, expand its capabilities to include global knowledge, and test its potential applicability and effectiveness by managing the complexity of a collection of autonomic systems in a case study of a major outage suffered by the Google Cloud Platform.**

***Index Terms*—Autonomic Systems, Hierarchical Control, Collections of Autonomic Systems, Meta-Management**

I. Introduction

Cloud providers have made the acquisition of additional compute capacity easily available at commodity pricing. Cloud providers provide some built-in autonomic capabilities into their services, that many applications take advantage of (e.g., auto-scaling). These capabilities harness the available compute capacity to maintain performance against defined quality objectives. Use of these readily available autonomic capabilities has caused the management complexity of the enterprise applications to increase. Each application is now a collection of autonomic systems in which each sub-system is an independent functional component.

The approach of combining autonomic sub-systems together to create a composite application can lead to behaviors that negatively impact the global aggregate utility of the application. For example, the operations management system for an enterprise cloud application is responsible for rolling out a security patch to all compute resources. In cloud environments, this is often achieved with a 'rolling restart' in which newly patched systems are brought online while unpatched systems are taken offline. At the same time, the auto-scaling capabilities for each sub-system are trying to maintain sufficient capacity to meet the quality objectives of the application. This leads to one system taking compute resources offline while another is trying to put them online. At least one of these actions could negatively impact the global aggregate utility of the application by either delaying the security update or impacting the ability of the application to meet quality objectives.

Commonly, human administrators address these conflicts as part of a design time analysis of the situation or a run time mitigation of the undesired effects. However, the task of controlling and mitigating undesirable emergent behaviors is complex and error prone. The administrator needs to analyze the current and potential future states of the global system and each sub-system operating in a unique environment with multiple types of uncertainty and multiple competing quality dimensions to select a mitigation plan from a combinatorially large set of possibilities.

In [9] we proposed an approach that addresses these problems by enabling the creation of a higher level autonomic system, referred to as a *meta-manager*. The key idea of the approach is to assume that the behavior of each autonomic subsystem is described by an adaptation policy; a representation of the adaptive actions an autonomic manager will deploy given a state of the environment and a state of a managed system. We also assume that each sub-system provides a set of parameters that allows the meta-manager to tune sub-system adaptation within a specified range of behaviors. This abstracts the heterogeneity of each autonomic sub-system to create a homogeneous representation of their adaptive behavior. The meta-manager can then exploit this homogeneity to synthesize a plan that determines the configuration settings of the subsystems most likely to improve global aggregate utility without subsuming the control functions nor directly orchestrating the actions of the autonomic managers across a collection of heterogeneous sub-systems.

As a practical exemplar of the challenges and consequences in the management of collections of autonomic systems, in this paper we apply our approach to the major outage of the network control plane that the Google Cloud Platform (GCP) suffered on June 2, 2019. The network control plane for GCP exhibits a multi-tiered autonomic system architecture

demonstrating a clear division of architectural responsibilities. Specifically, each of the network control plane clusters are independent and controlled by cluster management software which is responsible for appropriate auto-scaling activities and recovery or replacement of failed nodes within its specific cluster. Each of these independent clusters is, at least partially, controlled by a higher level autonomic manager and/or maintenance manager. This structural complexity combined with the criticality of the system and, specifically, its high availability requirements exemplify the challenges human administrators face in managing a collection of autonomic systems.

In this paper, we improve upon prior work with a more streamlined and understandable formal representation of the approach, expand its capabilities to include global knowledge, and test its potential applicability and effectiveness by managing the complexity of a collection of autonomic systems in a case study of a major outage suffered by the Google Cloud Platform (GCP) by answering two research questions:

1) **RQ1:** Would our automated approach have improved the performance of the network control plane in conditions similar to what GCP experienced?
2) **RQ2:** Will our automated approach scale sufficiently to meet the needs of an enterprise system similar to the one presented in the GCP case study?

Specifically, the contributions of this paper are:

1) An improved and more understandable formal representation of the approach and an expansion of its capabilities to include global knowledge in the management of collections of autonomic systems.
2) A point of validation of the effectiveness and applicability of our automated approach.

This paper is organized as follows: Section II provides the background on related work in relevant areas, Section III details the GCP outage including the architecture and corrective actions, Section IV discuses the automated approach to managing a collection of autonomic systems, Section V details the experiment demonstrating the effectiveness of the approach and results, Section VI presents the experimental results, and Section VII presents a discussion of current and potential future work.

II. Related Work

Automating the management of collections of autonomic systems was envisioned in the original paper defining the MAPE-K model for autonomic computing [12]. However, it is not entirely clear how one should do this. There are two readily available approaches; from control theory, a subsumption approach [1] [8] that replaces the autonomic management of the sub-systems with a higher-level manager that subsumes the control functions of the sub-system managers, and an orchestration approach [12] that updates the knowledge models of the sub-system managers with information relevant to their operation. Both of these approaches have a number of challenges.

The subsumption approach from control theory establishes hierarchical control systems by decomposing the complex behavior into individual units to divide the decision making responsibility. Each unit of the hierarchy is linked to a node in the tree and commands, tasks, and goals to be achieved flow down the tree from superior nodes, whereas sensations and commands results flow up the tree [1] [8]. This approach is effective in its application to systems with limited complexity. However, the complexity of the analysis necessary for a collection of autonomic systems would grow exponentially in the number of control actions and sub-system states making such a solution generally infeasible.

Since the orchestration approach was first elaborated by Kephart and Chess [12], there have been several efforts to address the scientific and engineering challenges of building these higher level autonomic managers specific to self-adaptive systems [18] [19]. An example of the scientific challenges, in [16], the authors develop a modelling language for collaborations in self-adaptive systems of systems. As an example of the engineering challenges, [14] develops a framework that defines a 'meta-model' that facilitates coordination between the different types of orchestrating autonomic managers. In [10], the authors create an autonomic sub-system that coordinates the activities necessary to facilitate network control plane communications. In [2], the authors create a meta-level control model to coordinate message between components in an agent based architecture using reinforcement learning. However, an underlying assumption of all this work is that by improving the coordination amongst the individual autonomic systems one can improve their performance against an implicit global utility function.

In [15] the authors combine the local utility functions into an explicit global utility function which is used to perform resource management. While the emphasis of the paper is on the use of utility functions in autonomic computing, the authors do create a higher level manager to handle a specific management function, resource allocation. While useful, this narrow focus will limit the applicability and effectiveness of the approach in modern computing environments that have diverse methods of measuring and improving global and local utility.

In [9] we introduced an approach that more broadly addresses the problem of modifying the adaptive behavior of the sub-systems so they make adaptation decisions that better align with an explicit global utility function. In the previous work we demonstrated the potential of the approach with a stochastic multi-player game (SMG) simulation. SMGs have been used to synthesize plans for individual autonomic systems [3], and our work extends that to synthesize plans for a collection of autonomic systems.

III. Google Case Study

On June 2, 2019, Google Cloud Platform (GCP) experienced a major outage in its scope, duration, and impact [11]. The outage caused network packet loss, up to 100%, resulting in an inability to access critical services for over 4 hours and caused a significant degradation of services for major websites including youtube.com, Gmail, GSuite, Nest, Snap,

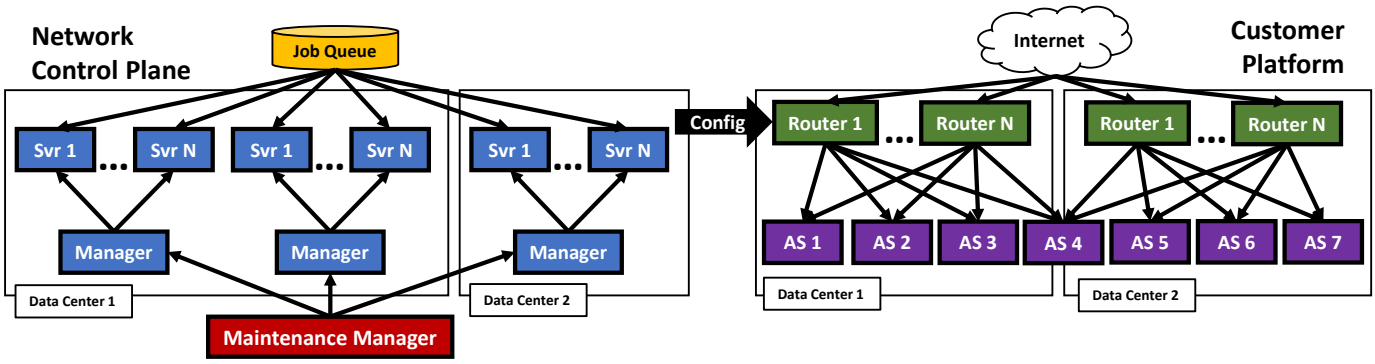

Fig. 1: GCP Control Plane Architecture

Discord, and Vimeo [13]. In this Section, we will detail, to the extent possible given public information, the architectural components and requirements of the system determined to be the root cause, how the outage progressed, the critical events during the remediation process, and a review of the preventive actions that resulted from Google's own post-mortem analysis.

A. Architecture

As is common with cloud providers, each customer has the ability to setup and customize a private networking space, known as an autonomous system (AS) or more commonly as a virtual private cloud (VPC), for their application. Due to the high degree of customization available, changes in the networking configuration of these VPCs are commonplace which mandates that a system is continuously processing these changes and making the relevant updates to the configuration of the physical networking hardware. This system is referred to as the network control plane.

While the technical details of the network control plane are not publicly available, Figure 1 presents what we believe to be a high-level and functionally accurate representation of the system based on the information provided in the outage details in [11]. The network control plane consists of a set of autonomous clusters composed of individual nodes and managed by cluster management software that enables various cluster control functions, like auto-scaling. Each of the nodes will process jobs to update the network configuration as needed. The exact mechanism behind this is unknown, but we have represented it as a queue. We would not expect this queueing system to be dedicated to the network control plane, but instead is an enterprise wide platform serving multiple applications.

These individual clusters are run under a specific GCP service known as Google Compute Engine [13] which leads us to believe the nodes are individual virtual machines. These individual clusters are distributed throughout various regions and availability zones to ensure a highly available and responsive network control plane. When the network control plane finishes processing a job, the new network configuration is then distributed to the relevant networking equipment to enable proper packet routing to the individual autonomous systems. Due to the expectations of customers and the criticality of changes in network configuration, the network control plane is constantly processing jobs maintaining very low processing time, anecdotally less than 5 seconds.

When maintenance of the network control plane is required, the maintenance system will either move jobs from one individual cluster to another or stop the processing of jobs and resume it after the maintenance on the cluster has been completed. In similar situations, maintenance is typically done in a 'rolling' fashion which allows one cluster to be under maintenance while others continue processing jobs, a strategy that prevents an interruption and degredation of service. Further, in the event of a failure of the entire network control plane, the physical network is setup to be 'fail static', meaning that the network will continue to run normally on the current known good configuration for a period of time to allow administrators to resolve the problem.

B. Outage Details

The GCP outage was the result of two misconfigurations and a software defect. Specifically, the network control plane jobs and their infrastructure were included in a specific type of automated maintenance event, the instances of the cluster management software were also included in the maintenance event type, and finally, the maintenance software had a defect allowing it to deschedule multiple independent software clusters at once, even if they were in different physical locations.

At 11:45 US/Pacific the maintenance event started and began to shutdown the clusters running the network control plane jobs. Once the network control plane failed, the physical network continued to operate normally for a few minutes. After this, the routing configuration was invalid resulting in significant packet loss, up to 100%, and end users began to be impacted between 11:47-11:49 US/Pacific. The Google engineers were alerted to the networking failures 2 minutes after it began and started their incident management protocols.

Troubleshooting of the failure was significantly hampered by severe network congestion by all of the consumers of GCP services which caused the engineering team to begin another set of incident management protocols to mitigate the tool failures. Specifically, engineers had to travel to the physical data centers, and they had to reprioritize network traffic to allow the tooling traffic to take precedence. By 13:01 US/Pacific, the root cause of the incident had been determined and the engineers began to re-enable the network control

plane and its supporting infrastructure. Due to the length of the outage and the shutdown of the network control plane instances across different physical locations, the configuration data had been lost and needed to be rebuilt. The rebuilt configuration for the network control plane began to roll out at 14:03 US/Pacific.

As the network control plane started to come back online, new network configurations began to roll out and service started to recover at 15:19 US/Pacific and full service was restored at 16:10 US/Pacific.

C. Post-Mortem Actions

Then GCP administrators implement a number of short term actions to prevent an immediate reoccurrence of the problem and plan a number of changes to prevent the problem from reoccurring. First, the administrators halt the datacenter automation software responsible for descheduling jobs for maintenance events. Second, they harden the cluster management software to reject requests to de-schedule jobs at multiple physical locations. Third, the network control plane will be reconfigured to handle the maintenance events correctly and persist its configuration so it will not need to be rebuilt. Finally, the GCP network will be updated to continue in a 'fail static' mode for a longer period of time in the event of a loss of the control plane [11].

IV. Automated Approach

Our approach, originally presented in [9], enables the creation of a meta-manager, which would assume control of many of the functions for managing a collection of autonomic systems typically performed by human administrators. In this work we simplify the formal representation of the model to improve understandability and extend the formal model to include additional information relevant to the meta-manager.

The key idea behind our automated approach to meta-management abstracts the heterogeneity of each sub-system by assuming that there exists a set of configuration options that can tune the behavior of the sub-system to within a specified range of behaviors and that the adaptive behaviors of each sub-system can be described by an adaptation policy. This creates a homogeneous representation of each sub-system that is exploited by the meta-manager to change the autonomic configuration options of the sub-system to improve the performance of the sub-system against a global objective.

These configuration options influence which adaptation tactics the autonomic manager selects to respond to events in the environment. For example, if a network control plane cluster at maximum configured server capacity determines that it does not have sufficient capacity to meet demand, it might choose to begin de-scheduling jobs. However, if the maximum server configuration were increased, then the cluster manager would have at least one additional adaptation tactic available, the ability to add additional servers.

The configuration options of an autonomic manager is one of, at least, three factors that can influence which adaptation tactic is employed. The second is the state of the environment. In the GCP example, this might be the number of network control plane jobs that are currently required to be processed. The third is the current state of the managed system. In the GCP example, this might be the current number of servers in use within a cluster or the processing power of each node. Once an adaptation tactic is applied, a new state of the managed system is the result. For example, the new state of a GCP control plane cluster might be an additional server over what was available in the previous state.

The configuration options, state of the managed system, and the state of the environment influence which adaptation tactics are potential candidates for selection, but the individual autonomic managers must attempt to select the 'best' adaptation action available. To do this, the adaptation manager evaluates the new state of the managed sub-system that results from the application of each adaptation tactic and determines its ability to meet the defined service level agreements (SLAs) of the system, often through the use of a utility function. The adaptation tactic that produces a state that maximizes the utility function of the sub-system is considered the 'best' adaptation tactic and is deployed.

Therefore, if one is able to enumerate the states of the managed system and the states of the environment, it is also possible to predetermine which adaptation tactic would be deployed by the autonomic manager given the state of the managed system and the state of the environment and subject to the configuration of the autonomic manager. This enumeration is referred to as the adaptation policy. In [9], the formal representation of the adaptation policy was built on a complex formal representation of a sub-system. A key difference in this work is that this formal representation is simplified by eliminating the model of the sub-system and instead represents the adaptation policy as a higher order function:

$$P(c) \rightarrow (p(e,s) \rightarrow s') \tag{1}$$

where:

- E is the set of all states of the environment that can be elaborated from the autonomic system environment model, $e \in E$, is a *state* as defined in [17]
- S is the set of all states of the managed system that can be elaborated from the autonomic system model, $s, s' \in S$, is a *state* as defined in [17]
- C is the set of all possible configurations and $c \in C$
- P is the adaptation policy of the adaptation manager it takes a configuration, c, and returns a function, p.
- $p(e,s)$ takes an environment state, e, and a system state, s, and returns the new system state, s'

The mechanism by which the autonomic manager moves from system state s to s' is through the use of an adaptation tactic, a, that is one of a set of adaptation tactics available, A where $a \in A$, and is subject to:

$$a \in \underset{i \in A}{\operatorname{argmax}} U(\lambda(s,i)) \tag{2}$$

where $U(s) \rightarrow [0..1]$ is the local utility function and $\lambda(s,a) \rightarrow s'$ is a transition function that returns the new state

of the managed system, s', given the current state, s, and the adaptation tactic, a.

Using the adaptation policy, P, it is possible to elaborate the extensive form of a game between the autonomic manager of the sub-system and its local environment. For example, for each time step, t, the local environment will establish a current state from those naturally possible, e_t, this will be the 'move' of the environment. Then, at the same time step, the sub-system will have a current state, s_t. Using e_t and s_t, one can then use the current configuration of the sub-system's autonomic manager, c, and the adaptation policy, $P(c) \rightarrow p(e_t, s_t)$, to determine which new system state, s', will be the result. This process can repeat for as many time steps and branches as necessary to elaborate the extensive form of the game. The extensive form of the game, as represented by the adaptation policy, for each sub-system is the only knowledge about the interworking of each sub-system the meta-manager has available.

The ability to elaborate an extensive form of the game between the sub-system and the local environment from the adaptation policy is what enables the meta-manager to analyze potential alternative configurations and for each sub-system and select the best configuration to improve their individual performance against a global objective over a time horizon. It is also what enables scalability of a meta-manager beyond trivial systems as the meta-manager does not subsume the control functions of each sub-system. Using the GCP control plane as an example, a meta-manager could analyze how a sub-system would adapt and perform against its objectives if the amount of available resources was reduced; a change to the sub-system's configuration. If it is found that the new level of resources would allow for performance that meets the SLAs of that sub-system, the meta-manager can then analyze the policies for the other sub-systems to see which might improve performance against SLAs if their available resources were increased. These changes in the configuration, or meta-tactics, of the sub-systems are one type of change a meta-manager could potentially make that allow the meta-manager to influence the adaptive behavior of the sub-systems without subsuming the autonomic control functions.

In addition to the adaptation policies, the meta-manager also has additional information, referred to as *global knowledge*, about the individual systems and the local environments that might be only partially known to the individual sub-systems. There are at least two types of global knowledge: 1) information on the interrelationships between local environments and 2) information on the interrelationships between sub-systems. Using the GCP control plane as an example, a cluster in one data center might be handling an unusually large number of jobs while another is near idle. The control plane will begin rescheduling jobs to the other cluster to balance the load. This is an example of one local environment of one sub-system having an impact on the local environment of another sub-system: the large load in one local environment triggered a more moderate load in another local environment.

Similarly, individual sub-systems can also have interrelationships between them. A common one is a shared resource pool, like money or energy, in which consumption of a resource by one sub-system can cause scarcity for the other sub-systems. Global knowledge allows the meta-manager to create adjusted representations of the states of the managed system and representations of the states of the environment for each of the sub-systems to enhance their accuracy making the meta-manager's analysis more effective.

We can formally define a meta-manager as a tuple $\mathcal{M}(\mathcal{U}, \mathcal{P}, \mathcal{C}, \mathcal{E}, \mathcal{S}, \mathcal{K})$ where:

- $\mathcal{U}$ is the global utility function where $\mathcal{U}(s) \rightarrow [0..1]$ where s is a managed system state,
- $\mathcal{P}$ is the set of all adaptation policies, P,
- $\mathcal{C}$ is the set of all sets of possible configurations for the subsystems, C,
- $\mathcal{E}$ is the set of all sets of possible states of the environment for the subsystems, E,
- $\mathcal{S}$ is the set of all sets of possible states of the managed system, S,
- $\mathcal{K}$ is a set of functions, $k(x) \rightarrow x' \in \mathcal{K}$, where x is a *state* as defined in [17]. Each function returns the new state that results from the application of the global knowledge represented by that function. It is important to note that both the states of the environment and states of the managed system derive from the *state* type in [17].

The goal of the meta-manager is to determine which configuration, $c \in C_i$, for a specific sub-system, i, will maximize the global utility function, $\mathcal{U}$, using a state of the environment, $e_i \in E_i$, a state of the managed system, $s_i \in S_i$, and the adaptation policy, $P_i \rightarrow p(e_i, s_i)$. Formally this can be stated as:

$$\forall P_i \in \mathcal{P}; \underset{c \in C_i}{\operatorname{argmax}} \mathcal{U}(P_i(c) \rightarrow p(e_i, s_i)) \tag{3}$$

where C_i is the set of all possible configurations for sub-system i, and e_i and s_i are the result of the application of global knowledge, $\mathcal{K}$:

$$e_i = \delta(\{\forall e \in E_i; \forall k(e) \in \mathcal{K}\}) \tag{4}$$

and

$$s_i = \theta(\{\forall s \in S_i; \forall k(s) \in \mathcal{K}\}) \tag{5}$$

$\delta(y) \rightarrow e$ and $\theta(z) \rightarrow s$ are functions that accept a set of states, environment(y) and managed system(z), and return the 'best' state to be used for the evaluation by the meta-manager. There is nothing about δ or θ which mandates a specific method of selection for the state of the environment or the state of the managed system. It is possible to select the current state, a state a specific number of time steps in the future, or even handle uncertainty to select the most likely state. This inclusion of the formal representation of global knowledge in our automated approach is a new contribution over the approach previously elaborated in [9].

An assumption built-in to our approach and formalization is that the global utility is only dependent upon the states of the managed systems that are part of the collection of autonomic systems under management. One limitation is that

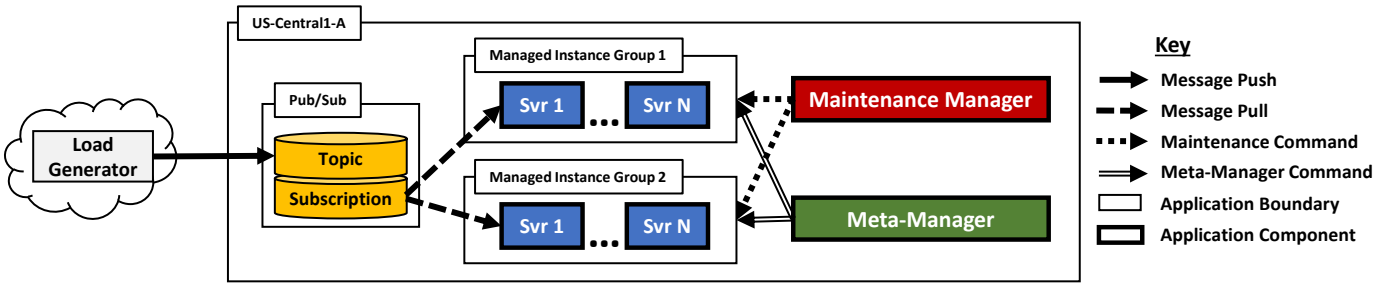

Fig. 2: Experiment Platform Architecture

this prohibits using attributes known only at the global level of the meta-manager itself.

We believe this to be a reasonable assumption because collection level factors that are useful in a utility calculation can be manifested in the managed system state of each sub-system. Returning to the GCP example, assuming there is a maximum operational run cost for the entire network control plane, the meta-manager can set and adjust the maximum cost levels, or number of servers, for each of the control plane clusters to prevent the summation of all of them from exceeding the defined maximum. The state of the managed system would then represent its level of cost which could then be factored into the global utility calculation.

A feature of our approach is that it does not mandate a specific type of analysis nor toolset to determine the best configuration for each sub-system. For example, one system implementation could use a rules-based approach while another uses a stochastic multi-player game approach; still others could use a non-exhaustive state space exploration method, like a Monte Carlo analysis. All of these analysis methods are enabled by the use of the adaptation policies. This is important for the GCP case study as it increases the potential scalability of the approach; making it more applicable to real collections of autonomic systems and making it a potentially effective automated alternative to a human-centric approach.

V. Experiment & Results

The goal of the experiment is to answer RQ1 and RQ2 and examine the applicability and effectiveness of our approach. Both research questions can be answered by creating a representative workload against a realistic experimental platform and evaluating three scenarios: normal conditions, maintenance conditions, and maintenance conditions with a meta-manager. To determine and compare the effectiveness of our automated approach, we use two measures: (1) the integral of the time of the oldest message in the queue, referred to as the *inverse utility*, and (2) the integral of the cost (i.e., number of servers in use per unit-time). The ideal value for each of these is the minimum that can be achieved. This appropriately evaluates the GCP case study as the control plane has the assumed quality attribute of processing the requested changes to the network configuration with minimal delay consistently over time; an inverse aggregate utility.

Additionally, we examine the scalability of the approach by exploiting the fact that each of the network control plane clusters are practically identical. This enables an assumption that the set of meta-tactics appropriate for one cluster is applicable to all clusters. We can then consolidate and simplify the analysis which increases the scalability of the approach. To test this, we re-ran the scenario with maintenance and a meta-manager to determine if there was a notable change in the aggregate utility as a result of applying the proposed model scaling technique.

A. Platform Implementation

The experiment was designed to mimic the GCP control plane architecture describe in Section III. Therefore, it was instantiated using GCP standard services.

The technical platform consists of five principal components, as diagrammed in Figure 2: (1) Load Generator, (2) Pub/Sub Queue, (3) Managed Instance Groups, (4) Maintenance Manager, and (5) the Meta-Manager.

The load generator is a custom utility using the Google SDK to place random well-formed messages on a Pub/Sub queue. The rate at which it places messages on the queue and the length of the run are both configurable.

The Pub/Sub queue is a standard GCP product offering which provides message queuing and subscriber capabilities with guaranteed delivery mechanisms. This was configured with a single topic and a single pull subscription.

A managed instance group (MIG) is a cluster of virtual machines (VMs) that are managed by an external controller that will auto-scale the cluster depending upon configured parameters. Each instance group is configured to auto-scale to maintain the Pub/Sub metric *oldest_unacked_message_age* less than or equal to 5 seconds. This metric represents the age of the oldest message in the Pub/Sub queue. Each instance group is also configured, by default, with a 60 second cool down time between auto-scale actions and a minimum of 1 VM and a maximum of 10. Each of these VM instances is created from a base operating system template running a custom utility that is configured to check for and, if present, pull messages from the Pub/Sub queue once every second. Processing of the messages is then simulated by determining a random amount of time, between 250 and 1500 milliseconds, the system pauses before discarding the message and moving to the next.

The maintenance manager is a custom developed utility running on a static virtual machine that uses the Google SDK to interface with the configuration and state of the MIGs.

Specifically, it can place the MIG into a 'Rolling Replace' update in which each VM in the cluster is replaced.

The meta-manager is a custom component that collects current state information from the managers of the MIGs, uses that information to update a model, presented in Section V-B, triggers the analysis of the model to determine the 'best' set of meta-tactics to deploy, and carries out those actions against the managed sub-systems.

B. Model Definition

The model used by the meta-manager is defined as a Stochastic Multi-player Game (SMG) that is implemented in PRISM-Games [6] v.2.1 using the PRISM language syntax. The meta-manager analyzes this to synthesize an adaptation strategy for the meta-manager to implement, see [3], [9], [4] for other examples of this technique. This strategy synthesis is first attempted during the experiment run-time using data from the running system. Parts of the model are labeled 'Dynamic' to indicate which information is injected at run time. However, if the analysis of the model cannot be completed in an experimentally relevant time horizon, the synthesis is performed off-line and loaded into the meta-manager for execution using default values.

1) Global Items: The items found in Listing 1 are a set of constants and global variables that facilitate the creation of the model. The variable *QueueLoad* is the current number of outstanding jobs to be processed. The variable *Model_Sink* is set to *True* to designate that the model is in its end state and prevents any further model actions. The constant *MAX_TURNS* sets the maximum number of steps, or time horizon, the model can run. The constants *MIN_SERVERS* and *MAX_SERVERS* set the bounds for the number of instances in the modeled MIG. The constant *PROCESSING_RATE* is the average number of new jobs each VM can process per unit time. The constant *NEW_JOBS_RATE* sets the average number of jobs per unit time. The constant *MAX_OLDEST_MSG_TIME* sets the threshold value for what is considered an acceptable processing time. The formula *CURRENT_CAPACITY* calculates the total amount of server capacity available. The formula *OLDEST_MSG_TIME* is dynamic and is replaced by the observed value of the *oldest_unacked_message_age* metric.

```
global QueueLoad : [0..10000] init 0;
global Model_Sink : bool init false ;

const int MAX_TURNS = 150; //DYNAMIC
const int MAX_SERVERS = 10;
const int MIN_SERVERS = 1;
const int PROCESSING_RATE = 150; //DYNAMIC
const int NEW_JOBS_RATE = 250; //DYNAMIC
const int MAX_OLDEST_MSG_TIME = 5;

formula CURRENT_CAPACITY = (MIG1_Server_Count) *
    PROCESSING_RATE;
formula OLDEST_MSG_TIME = (CURRENT_CAPACITY >=
    QueueLoad) ? 3 : 7; //DYNAMIC
```

Listing 1: Global Items

2) Reward Structures: The reward structures, combined with the properties (see Section V-B5), represent the global utility function, $\mathcal{U}$. This reward structure is setup to assign the same number of 'points' to each environment state as the time of the oldest message in the queue.

```
rewards "ControlPlane_Time"
  [AddJobs] true: OLDEST_MSG_TIME;
endrewards
```

Listing 2: Reward Structures

3) Control Module: The control module provides the administration for the model. Specifically, it tracks what the current turn is, *turn*, the turn count, *turnCount*, and a series of synchronized actions to update the current turn.

```
module ControlModule
  turn : [0..2] init 0;
  turnCount : [0..1000] init 0;
  [AddJobs] (!Model_Sink) -> (turn' = turn + 1) & (turnCount' =
      turnCount + 1);
  [ProcessJobs] (!Model_Sink) -> (turn' = turn + 1) & (turnCount'
      = turnCount + 1);
  [Maintenance] (!Model_Sink) -> (turn' = turn + 1) & (turnCount'
      = turnCount + 1);
  [AddCapacity] (!Model_Sink) -> (turn' = turn + 1) & (turnCount'
      = turnCount + 1);
  [RemoveCapacity] (!Model_Sink) -> (turn' = turn + 1) & (
      turnCount' = turnCount + 1);
  [CoolDown] (!Model_Sink) -> (turn' = turn + 1) & (turnCount' =
      turnCount + 1);
  [MM] (!Model_Sink) -> (turn' = 0) & (turnCount' = turnCount +
      1);
endmodule
```

Listing 3: Control Module

4) Player Definition: In our definition of the game, there are three players: (1) Environment, (2) MIG1, and (3) Meta-Manager. Listing 4 shows the model player definitions.

```
player ENV [AddJobs], Environment endplayer
player CONTROLPLANE [ProcessJobs], [Maintenance], [AddCapacity
    ], [RemoveCapacity],[CoolDown] endplayer
player MM [MM] endplayer
```

Listing 4: Player Definition

The first player is the environment which adds jobs to the queue and determines the end of the model by setting *Model_Sink*.

```
module Environment
  [AddJobs] (turn = 0) & (turnCount < MAX_TURNS) -> (
      QueueLoad' = QueueLoad + NEW_JOBS_RATE);
  [](turn = 0) & (turnCount >= MAX_TURNS) & (!Model_Sink)
      -> (Model_Sink' = true);
endmodule
```

Listing 5: Environment Definition

The second player is the MIG with 5 possible actions: (1) process jobs from the queue, (2) undertake maintenance, (3) add server capacity, (4) remove server capacity, and (5) cool down after adding or removing capacity. The actions within the module define the adaptation policy, P, for the sub-system, the MIG. To examine the *AddCapacity* action in detail, the statement *MIG1_Cool_Down_Count* and is an element from the managed system state, s, and statement *OLDEST_MSG_TIME*

```
const int MIG1_Turn = 1;
global MIG1_Server_Count : [MIN_SERVERS..MAX_SERVERS] init 1;
global MIG1_Cool_Down : [0..2] init 1;
global MIG1_Cool_Down_Count : [0..2] init 0;
global MIG1_CanMaintenance : bool init true;
formula MIG1_ProcessJobs = ((QueueLoad − (MIG1_Server_Count ∗ PROCESSING_RATE)) > 0) ? QueueLoad − (MIG1_Server_Count ∗ PROCESSING_RATE) : 0;
formula MIG1_SetMaintenanceSvrs = ((MIG1_Server_Count − 2) >= 1) ? MIG1_Server_Count − 2 : 1;
formula MIG1_AddCapacity = (MIG1_Server_Count + 1 > MAX_SERVERS) ? MAX_SERVERS : MIG1_Server_Count + 1;
formula MIG1_RemoveCapacity = (MIG1_Server_Count − 1 < MIN_SERVERS) ? MIN_SERVERS : MIG1_Server_Count − 1;

module MIG1
  [ProcessJobs] (turn = MIG1_Turn) −> (QueueLoad' = MIG1_ProcessJobs);
  [Maintenance] (turn = MIG1_Turn) & (MIG1_CanMaintenance) −> (MIG1_Server_Count' = MIG1_SetMaintenanceSvrs) & (MIG1_CanMaintenance' = false);
  [AddCapacity] (turn = MIG1_Turn) & (OLDEST_MSG_TIME > MAX_OLDEST_MSG_TIME) & (MIG1_Cool_Down_Count <= 0) −> (MIG1_Server_Count' = MIG1_AddCapacity) & (MIG1_Cool_Down_Count' = MIG1_Cool_Down);
  [RemoveCapacity] (turn = MIG1_Turn) & (OLDEST_MSG_TIME < MAX_OLDEST_MSG_TIME) & (MIG1_Cool_Down_Count <= 0) −> (MIG1_Server_Count' = MIG1_RemoveCapacity) & (MIG1_Cool_Down_Count' = MIG1_Cool_Down);
  [CoolDown] (turn = MIG1_Turn) & (MIG1_Cool_Down_Count > 0) −> (MIG1_Cool_Down_Count' = MIG1_Cool_Down_Count − 1);
endmodule
```

Listing 6: MIG Definition

> *MAX_OLDEST_MSG_TIME* is comparing an element from the system state to an element from the environment state, e. The result of this statement is an increase of +1 to the number of servers, up to the maximum, currently in the cluster. The declared global variables are all attributes of the configuration of the autonomic system and their state at any given point of time would be an entry $c \in C$ from Section IV.

The third player is the *MetaManager* module, presented in Listing 7, and it has a set of actions that represent the available meta-tactics. Each of these meta-tactics updates the configuration of *MIG1*. The guards on each of these actions are identical and each part of them is strictly to ensure the proper operation of the model, they do not influence which meta-tactic could be selected. This is important, as this is the uncertainty that will be resolved by the tool when it synthesizes a strategy to determine the best use of these meta-tactics; determining the adaptation strategy the meta-manager will use. This process is what is represented by equation

```
const int MM_Turn = 2;
module MetaManager
  [MM](turn = MM_Turn) −> (Model_Sink' = Model_Sink); //Pass−Through, No Action
  [MM](turn = MM_Turn) −> (MIG1_Cool_Down' = 0);
  [MM](turn = MM_Turn) −> (MIG1_Cool_Down' = 1);
  [MM](turn = MM_Turn) −> (MIG1_Cool_Down' = 2);
  [MM](turn = MM_Turn) −> (MIG1_CanMaintenance' = false);
  [MM](turn = MM_Turn) −> (MIG1_CanMaintenance' = true);
endmodule
```

Listing 7: MetaManager Definition

IV. The tool uses the current state of the environment, e_P, the current state of the managed system (*MIG1*), s_P, and the current configuration, c, to determine which configuration change is going to improve the global utility function, $\mathcal{U}$, see Section V-B2. The first statement is a pass-through with no effect on other components. This allows for the MetaManager to take 'no action'.

5) Properties: Listing 8 is a rewards based property that causes the tool to search for the strategy that minimizes the reward, see Section V-B2, that can be guaranteed by the players *MM* and *CONTROLPLANE*.

```
<<CONTROLPLANE,MM>> Rmin=? [ Fc Model_Sink ]
```

Listing 8: Properties

It is constructed using rPATL [5] [6] [7] which combines features of multi-agent logic ATL, the probabilistic logic PCTL, and operators to reason about different notions of cost/reward structures.

The *Model_Sink* is a property that is only true in the end state(s) of the SMG. This facilitates the calculation of the cost/reward for the SMG by providing a property that is common in every end state of the SMG.

C. Scenario Results

Each of these scenarios was run by generating a workload of 250 messages per minute for 30 consecutive minutes with an additional 5 minute warm-up and cool-down period against an enterprise production grade cloud system, not a simulation. All scenarios were run in a single 3 hr. window to control variability in the underlying systems.

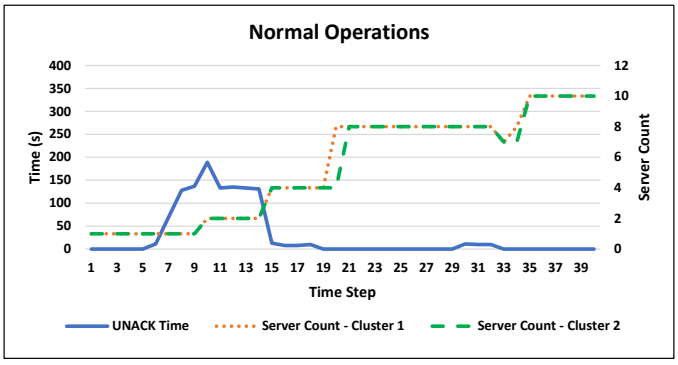

Fig. 3: Experiment - Normal Operations

The first scenario, presented in Figure 3, exercises the system under normal operating conditions without the interference of the maintenance manager or the assistance of a meta-manager. This scenario results in an aggregated inverse utility of 1137 with an aggregated cost of 431.

The second scenario, presented in Figure 4, exercises the system with the interference of a maintenance manager. The maintenance manager is configured to perform a rolling restart of cluster 1 at time step 10 and a rolling restart of cluster 2 at time step 15. This scenario results in an aggregated inverse utility of 3024 with an aggregated cost of 334.

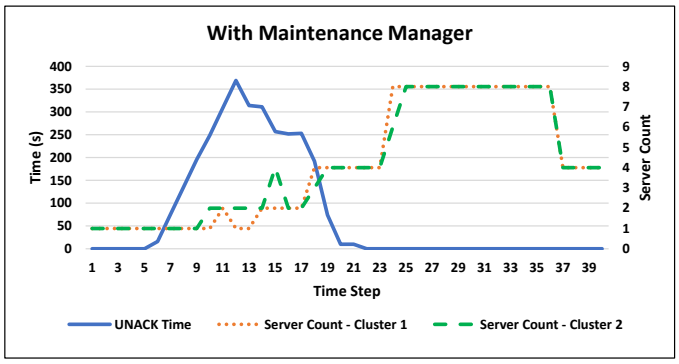

Fig. 4: Experiment - Maintenance Operations

The third scenario, presented in Figure 5, exercises the system with the interference of a maintenance manager, which is configured identically as the previous scenario, and the assistance of a meta-manager. In this scenario, the meta-manager first attempts to set the cool down period of each of the clusters from 60 seconds, the default, to the minimum available, 15 seconds at time step 7. Then at time step 12, the meta-manager configures each cluster to ignore the requests of the maintenance manager to perform rolling restarts. While the rolling restart has already been started for MIG1, this action does prevent the rolling restart of cluster 2 at time step 10. Due to the state space expansion of modelling both MIGs explicitly, it was necessary to reduce the *MAX_TURNS* to 40 which took a total of 642 seconds generating 12,299,356 states and 35,734,291 transitions. Because of the amount of time required to run this model, the meta-strategy was precalculated offline and preloaded into the meta-manager. This scenario results in an aggregated inverse utility of 1157 with an aggregated cost of 298.

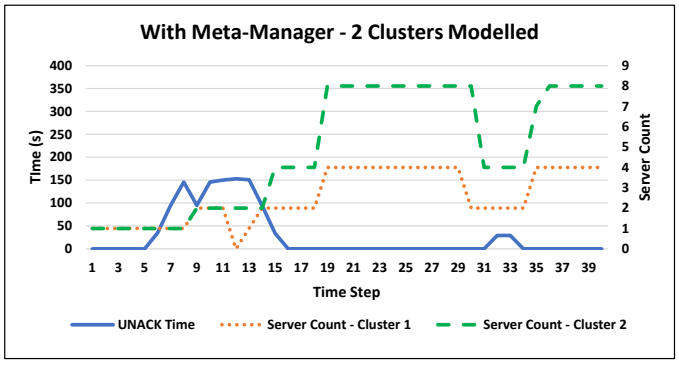

Fig. 5: Experiment - Meta-Manager Operations - 2 Clusters

Finally, the fourth scenario, presented in Figure 6, exercises the system identically to the third scenario. The difference being that the model used by the meta-manager has only one representative cluster, with two physical clusters, under the assumption that because the physical clusters are identical, the proposed meta-tactics appropriate for one would be applicable to all others. Similarly to scenario 3, the meta-manager reconfigures the cool down period for each cluster at time step 6 and configures each cluster to ignore the maintenance manager at time step 13, again preventing the rolling restart of cluster 2 at time step 15. The model took a total of 216 seconds to generate a strategy for the MetaManager generating 2,772,379 states with 6,840,217 transitions with a *MAX_TURNS* of 150. The scenario results in an aggregated inverse utility of 1095 with an aggregated cost of 269.

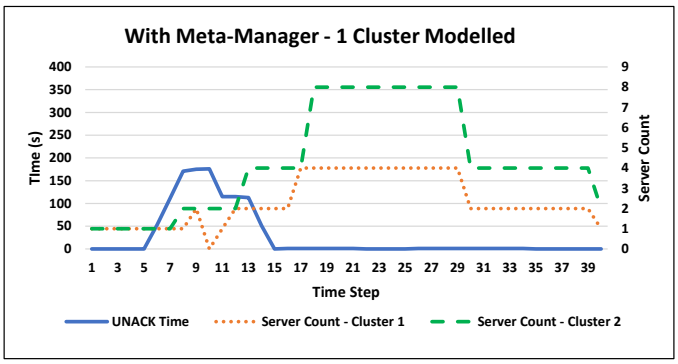

Fig. 6: Experiment - Meta-Manager Operations - 1 Cluster

VI. Experimental Results

The experimental results show a 61.7% improvement, from 3024 to 1157, in the aggregate inverse utility between scenario 2, maintenance conditions, and scenario 3, maintenance conditions with a meta-manager. Further, the results also show that the meta-managed scenario had a 10.7% improvement in the aggregated cost. Additionally, this is only a 1.75% difference between the normal operating conditions, scenario 1, and the meta-managed maintenance conditions, scenario 3. Based on these experimental results, we can reasonably conclude that our automated approach would have improved the performance of the network control plane in the GCP case study (RQ1).

Finally, the difference in aggregate inverse utility between the meta-managed scenarios, scenario 3, with 2 MIGs modeled, and scenario 4, was 5.4% decrease in aggregate inverse utility and 9.7% decrease in aggregated cost. While a 5.4% difference in the aggregated inverse utility could be significant in some contexts, we believe this difference is acceptable in a wide variety of contexts and can therefore conclude that our automated approach would scale sufficiently to meet the needs of an enterprise system similar to the one presented in the GCP case study (RQ2).

VII. Discussion & Future Work

The approach elaborated in Section IV is designed for the continuous improvement or self-optimization of a collection of autonomic systems, but it is not obvious that the GCP incident could potentially be mitigated by a self-optimization approach. The GCP Case Study in Section III highlights a set of specific faults and the recovery actions necessary. Self-optimization and fault detection and recovery are two distinct problems that define two different important lines of inquiry. However, as in this case, there are situations in which a self-optimization approach can determine that something is causing the cluster to behave sub-optimally and find a set of configurations that

could eliminate or partially mitigate the negative effects of the specific faults.

Any approach that synthesizes strategies is potentially challenged by the scalability of the solution. Our example demonstrated one potential method of enhancing scalability by exploiting the practical uniformity of all the sub-systems under management in the GCP network control plane. But there are additional ways of enhancing scalability as part of the individual analysis techniques. There is nothing about our approach that is specific to any off-the-shelf tool nor strategy synthesis technique. This partially enables our approach to address scalability challenges which addresses a degree of generality across a number of potential domains and types of collections of autonomic systems.

Future work in this area will focus on (1) better understanding the applicability of various strategy synthesis techniques and their applicability to various use cases based upon the level of assurance they provide and the timeliness of their analysis, (2) examining architecture strategies of composing autonomic systems into functional collections to evaluate the how the use of global knowledge and strategies to further enhance scalability differ and (3) loosening the goal of the collection of autonomic system of improving global aggregate utility and instead improve against a global objective. For example, in a security context, it might be in the collection's best interest to hold the attention of an attacker by sacrificing the currently compromised sub-system to allow time for the other members of the collection to mitigate the threat to themselves. This would be changing the priority of the meta-manager from improving global aggregate utility to improving performance against a global objective (e.g., maintaining security) which might be the desired short term objective.

ACKNOWLEDGMENTS

This work is supported in part by awards N000141612961 and N00014172899 from the Office of Naval Research. Any views, opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research.

REFERENCES

[1] James S. Albus. A reference model architecture for intelligent systems design. http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=820486. Accessed: 2019-10-18.

[2] Daniela Pereira Alves, Li Weigang, and Bueno Borges Souza. Using meta-level control with reinforcement learning to improve the performance of the agents. In Lipo Wang, Licheng Jiao, Guanming Shi, Xue Li, and Jing Liu, editors, *Fuzzy Systems and Knowledge Discovery*, pages 1109–1112, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] Javier Cámara, David Garlan, Bradley Schmerl, and Ashutosh Pandey. Optimal planning for architecture-based self-adaptation via model checking of stochastic games. In *Proceedings of the 10th DADS Track of the 30th ACM Symposium on Applied Computing*, Salamanca, Spain, 13-17 April 2015.

[4] Javier Cámara, Gabriel A. Moreno, David Garlan, and Bradley Schmerl. Analyzing latency-aware self-adaptation using stochastic games and simulations. *ACM Trans. Auton. Adapt. Syst.*, 10(4), January 2016.

[5] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. Automatic verification of competitive stochastic systems. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 315–330, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[6] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. Prism-games: A model checker for stochastic multi-player games. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 185–191, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, Aistis Simaitis, Ashutosh Trivedi, and Michael Ummels. Playing stochastic games precisely. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 – Concurrency Theory*, pages 348–363, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[8] W. Findeisen, F.N. Bailey, M. Brdys, K. Malinowski, P. Tatjewski, and A. Wozniak. *Control and Coordination in Hierarchical Systems*. International Series on Applied Systems Analysis. John Wiley & Sons, 1980.

[9] Thomas J. Glazier and David Garlan. An automated approach to the management of a collection of autonomic systems. In *Proceedings of the 4th eCAS Workshop on Engineering Collective Adaptive Systems*, Umea, Sweden, June 2019.

[10] Hemant Gogineni, Albert Greenberg, David A. Maltz, T. S. Eugene Ng, Hong Yan, and Hui Zhang. MMS: An autonomic network-layer foundation for network management. In *IEEE Journal on Selected Areas in Communications*, volume 28, pages 15–27. IEEE, 2010.

[11] Google. Google cloud networking incident #19009. https://status.cloud.google.com/incident/cloud-networking/19009. Accessed: 2019-10-12.

[12] Jeffrey Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36:41 – 50, 02 2003.

[13] Jonathan Shieber. Google cloud is down, affecting numerous applications and services. https://techcrunch.com/2019/06/02/google-cloud-is-down-affecting-numerous-applications-and-services/. Accessed: 2019-10-17.

[14] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. Model-driven architectural monitoring and adaptation for autonomic systems. In *International Conference on Autonomic Computing and Communications. ICAC '09.*, pages 67–68. ACM, 2009.

[15] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 70–77, 2004.

[16] Sebastian Wätzoldt and Holger Giese. Modeling collaborations in adaptive systems of systems. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15, pages 3:1–3:8, New York, NY, USA, 2015. ACM.

[17] Danny Weyns, Sam Malek, and Jesper Andersson. Forms: a formal reference model for self-adaptation. In *Proceedings of the 2010 International Conference on Autonomic Computing*, Washington D.C., USA, 2010.

[18] Danny Weyns, Sam Malek, and Jesper Andersson. On decentralized self-adaptation: lessons from the trenches and challenges for the future. In *SEAMS '10: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Cape Town, South Africa, May 2010.

[19] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl Goeschka. On patterns for decentralized dontrol in self-adaptive systems. In Rogério de Lemos, Holger Giese, Hausi Muller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*. Springer, 2012.