Expertise
and insight
for the future

Samuel Abney

Toward Self-Managing Microservices: Design of a Semi-Autonomous Data Collection Agent

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

21 September 2020

Metropolia
University of Applied Sciences

Author Title Number of Pages Date	Samuel Abney Toward Self-Managing Microservices: Design of a Semi-Autonomous Data Collection Agent 58 pages 21 September 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Smart Systems
Instructors	Sami Sainio, Senior Lecturer

As applications, networks and IT systems grow larger and increasingly complex, manual real-time management is often infeasible. As a solution to this problem, autonomous software has emerged as a key area for development. By following pre-configured rules or exercising artificial intelligence, programs can operate more independently, handling tasks such as monitoring logs, allocating system resources, and coordinating devices. This project details one such example, following the design and implementation of a semi-autonomous software agent and its impact as part of a larger data analysis and visualization platform.

Building on the basis of an existing data collection microservice written in Python, the project focuses on the addition of self-monitoring and self-healing features, such as automated state monitoring, configuration updates, and rule-based action handling. The project demonstrates how such features increase resilience, reliability and ease of use in both the service and the larger platform while ensuring that administrators and users retain the capabilities to fully control and customize the system. While noting the various challenges posed by automation, this project seeks to provide evidence that, in many cases, its enormous benefits clearly outweigh its apparent costs.

Keywords	agent, automation, autonomous, autonomy, data, microservices

Contents

Acknowledgements

1 Introduction 1

2 Theoretical Background 4

2.1 The Choice of a Microservices Architecture 4
2.1.1 Difference from a Monolithic Architecture 6
2.1.2 Relation to a Service-Oriented Architecture 7
2.1.3 Advantages and Disadvantages 8
2.2 Autonomy in Software Design 10
2.2.1 Defining an Autonomous System 10
2.2.2 Advantages Enabled by Automation 11
2.2.3 Capabilities of Autonomous Systems 12
2.2.4 Approaching Autonomous Microservices 14
2.3 Defining the Data Collection Agent 15
2.3.1 The Concept of a Software Agent 15
2.3.2 Challenges of Agent Design 17
2.3.3 The Data Collection Agent 18
2.4 System Context and Architecture 19
2.5 Intended Contributions of this Project 23

3 Methods and Implementation 24

3.1 Defining Implementation Aims 24
3.2 Tools and Technologies 25
3.2.1 Programming Language 25
3.2.2 Configuration File Format 26
3.2.3 Service Process Management 27
3.2.4 Testing Setup 29
3.3 Choosing an Implementation Strategy 29
3.4 Conceptualizing the Service Daemon 31
3.5 Communication Roles and Interfaces 32
3.6 The Service Daemon Configuration File 37
3.7 State Monitoring and the Integrated DCA 41
3.8 Action Handling 44
3.9 Configuration Updates 44
3.10 Monitoring DCA Service Status 46

3.11 Service Installation and Initialization 47

4 Results 48

5 Discussion 50

5.1 Remaining Challenges and Potential for Improvement 50

5.2 Future Expansion 51

6 Conclusion 54

References 56

Acknowledgements

I would like to thank Juri Sipilä and Convergens Oy for the opportunity to work on the system that forms the basis of this project and for the support, input, and feedback that helped me achieve a stronger result. I would also like to thank my coworker Martin Johanson, with whom I have worked collaboratively on the system from the start and whose ideas and contributions helped to make this project possible.

1 Introduction

The introduction of autonomous control mechanisms is among the most important trends in software development today, as confirmed by even a cursory glance through popular technology news sources [1; 2; 3]. Both in small-scale, independent projects and major product lines from the largest corporations, there is a clear movement toward increasing productivity and ease of use by having software itself handle tasks that are tedious, repetitious, inconvenient, or difficult for human users.

Further development of autonomous systems brings with it a variety of challenges. The prospect of self-driving vehicles, for example, threatens to disrupt traditional work patterns while also challenging norms and expectations about transportation [4]. Groundbreaking autonomous diagnostic devices have begun to outperform human physicians in some respects, leading some to question whether they might render some medical specializations obsolete [5]. While the benefits appear inarguable, such examples help to explain the discomfort some feel about automation.

At the same time, less controversial examples can readily be found, where the benefits of autonomy clearly outweigh the apparent costs, making software and the systems that run it more approachable, intuitive, and useful for users. Examples range from simple desktop automation, which saves time and effort while remaining largely user-directed, to more complex business and industrial applications—such as scheduled maintenance tasks or monitoring facilities for unsafe conditions—where there is little direct input, yet, rather than replacing human administrators, autonomous systems can actually enhance their insight into, and control over, their operations.

This project draws inspiration from this important trend by adding semi-autonomous elements to the control mechanisms of a data collection agent within an existing microservice-based, event-driven system. By adding these features, the project aims to further the system's original goals and planned use cases, offering an improved experience and greater value to both system administrators and non-technical users.

The work done for this project is part of a larger collaborative effort over the course of a year building a complete end-to-end system for data analysis and visualization. This system has been designed with the aim of meeting the needs of a diverse set of customers

who would benefit from a platform for easy, intuitive, simultaneous monitoring of processes. These processes could involve anything from large-scale business operations, to hardware health monitoring, to tracking sensor data or devices used by individuals.

The goal of creating the system was in part to offer a service to customers by establishing a complete solution for data monitoring, built partly on freely available open-source components, but which addresses needs for which no ready-made solution exists and leverages its components in a cohesive way which would be prohibitively difficult to realize for a non-technical user and would require resources to run and maintain beyond what many customers would be able to invest.

The architecture of this system has also been planned from the beginning to allow for scalability for use by organizations of different sizes and to provide numerous points of customization and tailoring to the particular needs of the customer. This project focuses on one aspect of addressing those customer needs—the addition of autonomous behaviors at the point of data collection, part of a more general effort to add these attributes throughout the system. Table 1 below describes the targeted behaviors and their planned use cases.

Table 1. Desired Semi-Autonomous Behaviors [adapted from 16, p. 44]

Behavior Type	Implementation Goal	Administrator/User Benefit
Self-monitoring	A system service can evaluate its own states and those of its hardware, note (and notify users of) important events, and detect/predict its own failure.	- Administrators: Less need for constant vigilance in case of failure. - Users: Less need to monitor the user interface.
Self-healing	A system service can take action to repair/restart itself or its parts when it experiences a failure.	- Administrators: Less call for manual restarting, resetting, and other fixes. - Both: Less downtime.
Self-updating	A system service can alter its own configuration and test any changes made.	- Both: Less need for manual configuration.
Self-maintenance	A system service can keep itself in a working state and take/request actions that ensure it continues to perform well.	- Administrators: Less need for manual maintenance. - Both: Increased reliability and availability. Less downtime.

As the table indicates, the goal of adding semi-autonomous control to the system extends to all of its component services and hardware resources. The principle of reusability, often cited as key to the microservices architecture on which the system is based [e.g.

6], suggests the utility of a common service daemon that can be configured to manage the resources and features of any service to which it is assigned. For the purposes of this project, however, it is useful to limit the scope to the case of one service, the data collection agent (DCA).

The DCA operates at the data’s point of origin, and it features a modular structure with a library of data collection modules whose instances can be added or removed according to the desired configuration. The DCA is a particularly interesting focal point for this project, as it executes an array of processes to be monitored, produces varied types of event data with some local processing, and involves a variety of hardware, including the nodes on which the service runs and the various sensors it may use. By enhancing the DCA with the potential for semi-autonomous control, this project will explore some of the principles of system design and possibilities for automation to extend the system’s existing capabilities and more fully realize its foundational goals.

2 Theoretical Background

This project and the larger system it aims to enhance are based on principles of program design that are products of an extensive history of research and development. These principles continue to play central roles in ongoing developments in software engineering today, and their impact is felt across a range of industries, as developers seek to create systems ready for the future—one where software is more accessible and accomplishes more with less direction, where the availability and integrity of data are more important than ever before, and where intelligent assistants extend human capabilities.

One encapsulation of this coming future frequently mentioned throughout the literature is the notion of an emerging fourth industrial revolution, commonly known under the German name “Industrie 4.0.” This revolution is predicted to focus centrally on use of “smart” technologies in production, founded on such principles as “decentralization, … service orientation, and modularity” [7, pp. 3, 5]. These timely topics are all centrally involved in the design of a semi-autonomous data collection agent.

While many of the topics this project touches on, such as autonomy and decentralization, are undoubtedly wide-ranging, they can be distilled down to a few points that are particularly important as background. These include the choice of a microservices architecture, uses of intelligent software agents, and the theory and applications of autonomous systems with features of self-awareness.

In addition to the discussion of the theory and general applications of these topics, the relation of this project to the preceding work of building the larger system will be presented. An overview of the current state of the existing system, how it implements these concepts, and the role of the DCA as one piece of the larger system will provide important context and also help to elucidate the motivation for the project. Finally, given all of this background information, the project’s intended contribution to the field will be discussed.

2.1 The Choice of a Microservices Architecture

Widely adopted in recent years, a Microservices Architecture (MSA) is commonly described as a way of breaking down the organization of an application according to the specific roles its parts play. To put it in other terms, MSA is considered by some to be a

form of “modularization of software,” where the modularity achieved is not merely conceptual but functional as well [8, ch. 1]. While different definitions emphasize different aspects of what microservices are, an overview of frequently cited qualities they exhibit is sufficient to understand their aim.

The first of these key qualities relates to the size and scope of processes. Systems exhibiting MSA are built on functional units best described as “small” (in their domain of activity), or narrowly focused, which run separately and relatively independently, being connected only by “lightweight” channels of communication [9, p. 6569]. As a result, these systems benefit in a variety of ways from the robustness, potential for fine-tuning and service-specific scalability and replaceability MSA enables [10, pp. 5, 8].

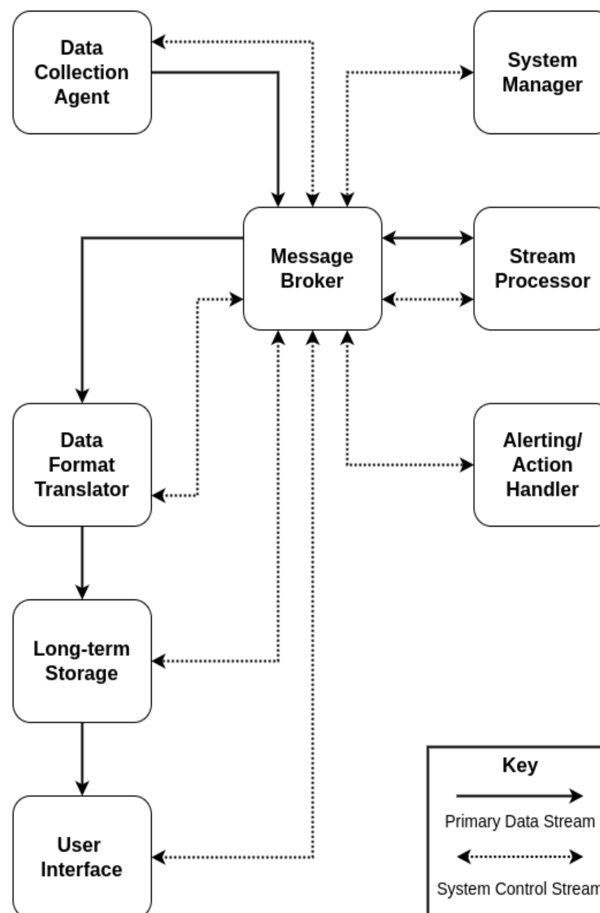

Figure 1. A simplified system diagram. The system’s microservices work closely together while remaining relatively independent.

These characteristics are illustrated in the simplified diagram included as figure 1 above, which depicts the system under development during the present project. The system is

characterized by a variety of small, independently operating microservices that are tied together by lines of communication, including both a stream of collected data and a stream of control signals, through which services can affect each other's operation.

Wolff points out that the concept of the individual microservice itself has no universally agreed-upon definition [8, ch. 1]. Still, the philosophical underpinnings of MSA are fairly evident. Some, including Wolff, connect the concept to the UNIX philosophy, which holds that programs should be small, self-contained units that do exactly one thing well and are intended from the start to be used in combination to accomplish more complex tasks [8, ch. 1]. In much the same way, microservices typically do something simple on their own but do complicated work when used together as a system.

2.1.1 Difference from a Monolithic Architecture

To fully define what MSA is, it is necessary to characterize what distinguishes it from competing architectures. Of these, the clearest contrast can be seen between MSA and the so-called monolithic structure of traditional applications, where all functionality is contained in a single large process. Mikkelsen et al. characterize the contrast between these two architectures especially in terms of scalability and the ability to quickly implement changes to functional parts of an application or system [9, p. 6569].

Restating this distinction, one can say that monolithic applications have a definite, preconceived structure and design that does not allow for a great deal of flexibility. Parts cannot be tested or modified independently, as they are inseparable from the whole application. Figure 2 below illustrates how the system depicted in figure 1 might look if conceived as a monolithic application. Rather than small, independent services, the single program is made up of corresponding layers that handle data in a definite, predefined process that cannot be modified without significant new development efforts.

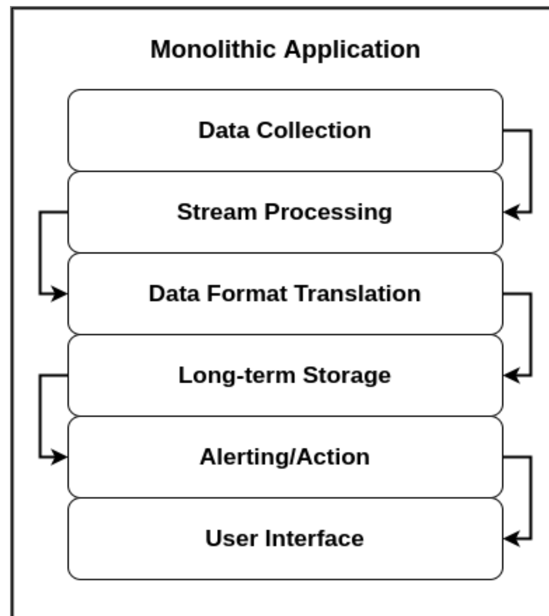

Figure 2. Conceived as a monolith, the system becomes a single, multi-layered application.

Microservice-based applications, in contrast with their monolithic counterparts, are very flexible. They are designed from the beginning to be adaptable to a wide variety of environments and use cases, and they are less fixed, because they are intended to be reconfigurable and upgradeable without the degree of difficulty that comes with a monolithic application. Because they are separate entities, services can be tested and replaced individually at any time, and the communication relationships between them can be modified with relative ease, which is a compelling advantage that saves development time.

2.1.2 Relation to a Service-Oriented Architecture

Numerous sources describe a debate over MSA's place as either a further development of, or a differently focused alternative to, a Service-Oriented Architecture (SOA) [e.g. 6]. SOA is an earlier approach to application design that shares MSA's focus on how parts function and what they can offer, in contrast with a monolithic architecture.

Mikkelsen et al. note that monolithic architectures may exhibit a "service orientation" where the SOA/MSA features of "high cohesion and low coupling" (in other words, independence of services which nonetheless function well as parts of a whole) are realized on a purely conceptual level in logically distinct functional parts of an application [9, p. 6569]. Yet this architecture leaves systems vulnerable to being taken down by even small

bugs, as the conceptual separation is not fully realized in practice. This fact motivated the development of alternatives such as SOA.

Baresi et al. describe SOA as a kind of ecosystem of “point-to-point interactions” among components and clients, where the former “export the services they provide,” while the latter are then able to find services that offer the capabilities they need [11, p. 27]. This is a key commonality with MSA, which may be considered simply to provide smaller, more focused services. According to Clark, however, SOA is often considered to be about exposing application services for use by other applications in an enterprise setting; with this understanding, it would be clear that MSA, which pertains to the internal structure of a single application, is something fundamentally distinct [6].

Following this line of thought, one could conclude that a defining difference in MSA is that it is the architecture most focused on functional independence. As Clark puts it, this is manifested in a few key ways: MSA avoids true reuse of components, instead preferring “reuse by copy,” it prefers asynchronous communication to reduce dependencies, it suggests a “discovery” model based on the assumption of volatile service availability, and it relies upon redundant copies and different “views” of data within separate services [6].

2.1.3 Advantages and Disadvantages

The choice of an MSA provides a variety of benefits compared with competing architectures, while also introducing some potential downsides. Several of these have particular bearing on the system in development, and even the possible disadvantages add motivation for the addition of autonomous features.

The robustness of MSA-based systems is evident in a variety of ways. In the conceptual and practical separation of their services, these systems distribute risk in a way competing architectures cannot. In a monolithic structure, a failure anywhere necessarily means a failure of the entire system, but the isolation of issues in microservices ideally averts the issue of “cascading failures,” where one small problem multiplies into other, larger ones, without affecting the functioning of the whole system [12, p. 3; 10, p. 5].

MSA systems are also designed such that their constituent services are easy to update and replace, avoiding the problem of aging systems in need of a full overhaul, which can

dissuade companies from making needed improvements [10, p. 7]. Changes to any particular microservice can be deployed without updating an entire system at once [12, p. 3]. This reduces the frequency of "large-impact, high-risk" deployments that are often avoided, leading to a continued accumulation of changes that will ultimately make a deployment even riskier [10, p. 6].

In a monolithic application, all parts necessarily change together. Just as an update is an update to the whole application, when the application needs to scale to handle more data, for example, it can only do so uniformly. This is particularly frustrating when the change in scale is only necessary because of performance constraints on a small piece of the much larger system [10, p. 5].

This highlights another key advantage of MSA. In contrast to competing architectures, it inherently provides the possibility of "fine-grained scalability," allowing different services to run at different scales, depending on their requirements, and ensuring that the scale at which one service is deployed can change even as another service remains as it was [12, p. 3]. Services that are less resource-intensive can run comfortably on low-end hardware while available resources are used where they are actually needed [10, pp. 5-6].

Developers also benefit from the MSA approach, exercising increased control over how they work. By enabling what is variously called "technological heterogeneity" or "polyglot programming," MSA gives developers the freedom to use languages and tools that they feel comfortable with or that prove to be the correct choices for what they wish to accomplish [12, p. 3].

This freedom of choice for developers is an important advantage of MSA compared to monolithic development, where developers may be forced to use a tool simply because it was selected for some previous implementation or forced to choose a tool that makes some tasks more difficult because it is "more standardized" or more applicable to all the different parts of a system [10, p. 4]. This kind of limitation can turn into a substantial handicap or lead to additional work and unforeseen problems.

The additional challenges MSA introduces are inextricably linked to many of its advantages. From the very beginning, the design of an MSA can itself be complicated, as it may not be obvious how to break down an existing application or organize a new system. Hassan and Bahsoon point to the challenge of determining an "optimal level of

granularity" for services—in other words, choosing how small services should be, what functional limitations they should have, and how many services there should be in a given system [13, p. 813]. There are no hard and fast rules to answer these questions.

Design is not the only challenge, however. Because microservices, by their nature, form a distributed system, they increase the need for mechanisms to ensure their effective cooperation and monitor their resource usage and functioning [12, p. 3]. This points to the potential benefits of self-aware services that can reduce the need for administrators to monitor services manually.

In addition to this, distributed services greatly increase the demands for testing and deployment [10, p. 11]. They also generate additional problems related to data storage and retrieval and ensuring working communications across more complex networks, all contributing to an increase in the number of "potential failure points" the system "exposes" and the difficulty of determining both the causes of and optimal resolutions to failures [12, p. 3]. These challenges, too, can ultimately be seen as motivating increased use of automation to mitigate this increasing complexity.

2.2 Autonomy in Software Design

A second major topic providing essential context for this project is the state of autonomous software. Even more so than in the case of MSA, autonomy and automation are the central focus of many major developments in software today, from smartphone applications and home assistants to self-driving cars and beyond. The topic is of such currency and widespread interest that a truly comprehensive summary of its state is beyond the scope of this project. It will suffice instead to provide a brief overview of its conceptual foundations and a selection of relevant applications.

2.2.1 Defining an Autonomous System

Generally speaking, autonomy can be associated with such concepts as self-determination and decision-making abilities. One may connect it to the idea of free will, thinking of a human being's exercise of personal autonomy. These connections are just as valid in the context of autonomous software, which is also intended to act independently.

An exact definition of what an autonomous system does proves more elusive, though all such systems aim at embodying the same general concepts. In the context of software agents, for example, Unland defines autonomy as possessing “sole control” of an agent’s own “internal state and … goals” [14, p. 4]. This expresses the core ideas of self-management and freedom from intervention.

The scope of systems and applications that might be described as fully or partially “autonomous” is quite wide, including everything from simple automation of tasks to fully independent intelligent agents. King identifies five distinct types of automation with varying degrees of autonomy, ranging from a user-controlled variety, such as desktop or device automation, to true artificial intelligence [15, p. 15]. Somewhere between these extremes lies what King refers to as “unattended automation,” distinguished from the former by the lack of human input and from the latter by being programmed directly to accomplish clearly defined tasks [15, p. 19].

This middle unattended level, which one might also relate to a “semi-autonomous” process in its being quite independent yet subject to restrictions, is most interesting for applications where the desired behaviors and abilities are defined in advance, as in this project. As King puts it, such a process runs on its own as if “in a black box,” all the while “interacting with IT systems” in more or less the same fashion as a “real user” [15, pp. 19-20]. In this sense, unattended automation describes quite well the desired role of a semi-autonomous data collection agent.

2.2.2 Advantages Enabled by Automation

The primary reasons for adding automation to software are straightforward—these features make common tasks easier and faster and often do things that humans cannot or achieve super-human efficiency. As the system under study in this project scales for a larger deployment, for example, log entries will accumulate at rates that even a large team could not effectively monitor in real time. Autonomous monitoring is therefore the most effective way to catch emerging problems before they escalate into more serious situations. By letting the software do as much work as possible, automation reduces unnecessary expenditures of time and effort. At the same time, it provides the added benefit of removing opportunities for human interference or user error.

King ascribes a variety of advantages to the various forms of automated processes, all generally expanding on the core features mentioned above. Reducing the effort demanded of users also promotes efficiency and results in systems that scale more easily; reducing the potential for errors made by users mitigates risk, makes behaviors more predictable, and ensures that best practices are followed [15, pp. 38-39]. These benefits grow with increasing levels of automation.

At the level of unattended automation more specifically, one can undoubtedly see how this is the case. Eliminating "messy" user input to the greatest extent possible brings processes even more under the control of the developer. At the same time, implementing this control requires a developer to become good at foreseeing problems. This is because all possible exceptions need to be handled in code to keep the system running smoothly on its own [15, p. 19]. In the end, developers must weigh the time and effort needed to implement autonomous features against the time they save and the gains they enable.

2.2.3 Capabilities of Autonomous Systems

The specific features and capabilities of autonomous systems vary widely, but they generally fall into a few categories. Lemoine et al. highlight what they see as the definitive properties of autonomous software agents, all of which relate in different ways to their capacity for degrees of awareness of their own environments and states.

Among other characteristics, autonomous agents may typically be described as "self-defining," "self-assembling," and "self-adapting," and they possess such capabilities as "self-control," "self-diagnosis," "self-simulation" and "self-repair" [16, p. 44]. Any of these properties could be relevant to and desirable in the case of a semi-autonomous DCA. As a more specific focus of this project, however, an agent's abilities to monitor and heal itself are of particular interest, as are practical examples that implement these properties.

Self-monitoring Examples

A self-monitoring autonomous system can take many forms. In a system that collects data, one important aspect of self-monitoring can be data verification, where unexpected measurement deviations can be a useful diagnostic for determining that a system component may have malfunctioned. Scully-Allison et al. detail the design of an autonomous

system for near-real-time monitoring of data for quality control purposes. Looking for errors in time-series data, including inconsistencies seen among sensors in the same locations, and data that is missing, repeated, or outside of defined bounds, the system they develop records metadata to mark failures and note which criteria of invalidity were met for a given sensor at a given time [17, p. 1657].

Monitoring a system's own functioning has applications beyond simple data checking, however. Koch and Pauls outline the implementation of an autonomous self-protection mechanism for monitoring access control. Using the example of a calendar system, they detail methods for the system itself to protect data by applying rules in response to function calls that verify and modify user rights, ensuring that data can only be changed or deleted by its owners [18, pp. 41-42].

One of the most important applications of self-monitoring is in diagnosing basic functional failures of the application itself or the hardware on which it is running. While outward signs like invalid data can indicate a deeper issue, such as a failed sensor, direct monitoring of resources and processes is also essential, in simple systems running on desktop hardware and even more so in complex applications such as self-driving vehicles, where failures could be catastrophic.

Self-healing Examples

When failures do inevitably occur, another key feature of autonomous systems is an ability to self-heal. This can range from simply restarting sub-processes or hardware all the way to complex reconfiguration or reorganization of systems. Baresi et al. propose a model for a self-healing system based on an SOA. Using the example of a pizza ordering and delivery process, involving a sequence of calls to various services, they imagine various failure scenarios in which services become unavailable, produce exceptions, or otherwise fail to provide needed information [11, pp. 29-31]. In such cases, they argue, the most resilient system would be able to circumvent problematic services on its own.

They suggest a toolbox of varied approaches for reacting to issues, including strategies to "re-try" interactions with the same service, "re-bind" to other services offering needed functionality, and "re-organize" to utilize combinations of otherwise incompatible services to achieve the same results [11, p. 38]. Ultimately, ordered combinations of simple strategies could provide more sophisticated response algorithms [11, p. 39].

Another example of the practical application of self-healing properties is Rajagopalan and Jamjoom's project app-bisect, which seeks to implement autonomous testing and rollback of configurations. Intended for use with microservices, app-bisect represents updates to services as a graph of "mutations," which, in the event of a failure, can be navigated in reverse as a treelike "changelog" [19, p. 1]. By running alternate systems constituted of different service versions in parallel, the program can look for a "least destructive combination of versions" that is preserved as the new production system pending administrator intervention [19, p. 1].

2.2.4 Approaching Autonomous Microservices

A number of the examples in the preceding sections anticipate a further important synthesis of the ideas presented above, the combination of MSA and autonomy and the additional benefits that accrue from their use in the same system. The app-bisect project, in particular, recognizes the potential power of applying autonomous control to managing microservices.

In his book Building Microservices, Newman actually defines microservices in terms of their "autonomy," referring primarily to the importance of their functional independence [10, pp. 2-3]. Mikkelsen et al. similarly frame their defense of four essential design principles for MSA in terms of their application to "Autonomous Microservices," even including the term in their title [9, p. 6571]. It is clear that bringing greater autonomy into systems built on microservices represents a greater fulfillment of the MSA philosophy.

In a practical application of this fusion, similar to the SOA-focused approach mentioned above, Lemoine et al. imagine a self-organizing system of microservice-based IoT devices, in which system-level functionality emerges without the need for direct configuration by users. As they see it, in future IoT applications, realized as "service compositions" with specific requirements, devices could enter or leave systems, which then reconfigure themselves by processes of discovering the presence and service offerings of other nearby devices [16, pp. 41-43].

From another angle, Hassan and Bahsoon envisage autonomous software as a possible solution even to MSA design difficulties. They propose that a feedback loop combining system monitoring with continual adjustments to the architecture could be used to create

a self-adaptive MSA that could use insights gained by running in production to improve upon the initial designs of system architects [13, pp. 813, 816].

Such examples show the variety of possibilities for enabling autonomous behaviors in MSA-based systems. What is also clear is that autonomous software and MSA share much in common in terms of basic principles and have great potential to complement each other in practical application.

2.3 Defining the Data Collection Agent

The DCA service that is this project's main focus is, as its name suggests, a type of software agent. As mentioned above, the term "agent" is frequently used in the context of autonomous systems. Therefore, a fuller discussion of agents in general, as well as the specific implementation details of the DCA, will clarify its role and the project's goals.

2.3.1 The Concept of a Software Agent

Software agents exemplify the application of autonomous behaviors. As long ago as 1997, when many current realities of automation were still only imagined in science fiction, Shoham and Bradshaw defined software agents as autonomous entities exhibiting intelligence and flexibility, carrying out their tasks in environments that they can react to and learn from, alongside other such entities [20, p. 7].

While the possibilities for implementing these agents have expanded dramatically since that time, the basic concept of what they should be has remained basically unchanged. Unland defines agents in much the same terms. They are "autonomous, problem-solving and goal-driven," pursue their goals by "proactive" means, and maintain awareness of their "dynamic environment[s]" [14, p. 3].

What this means in practice varies greatly, as the degree and type of autonomous behavior varies. Ribeiro notes that the term "agent" has been used variously to refer to anything from a UNIX daemon or algorithm to an "embodied agent" such as those found in robotics [21, p. 45]. Agents vary, therefore, not only in sophistication but also in their capability to manipulate the physical world.

In today's world, agency in software is gaining new associations. The term "agent" may bring to mind "conversational agents" such as chatbots and voice assistants, now becoming ubiquitous and part of daily life for average consumers [22, pp. 1-2]. One may also think of the programs driving the autonomous vehicles now under development. These agents represent different types that vary greatly in intelligence and features.

Agents can be categorized into types along a spectrum of behaviors corresponding to different degrees of autonomy. Unland characterizes a range of agents from those that are merely "reactive" to the fully "deliberative" variety [14, p. 4]. Reactive agents exhibit a much simpler design, focusing on responsiveness to environmental stimuli rather than more complicated internal modeling of their environments [14, p. 5]. Their intelligence is limited to an ability to act according to set guidelines. They are optimized for pattern matching and act on the basis of configured "situation-action associations" [14, p. 5].

In this way, reactive agents are relatively limited by the fixed constraints of their design. They prioritize speed and conservation of resources over flexibility and "dynamic" updating [14, p. 5]. Yet these limitations and priorities make them well suited to many scenarios.

Deliberative agents, on the other hand, are "intentional," "proactive" and "sophisticated," can be said to have "comprehensive real-world knowledge" of some narrowly focused topic related to the goals they are tasked with achieving, and behave in a way that is "neither fully explainable nor deterministic" [14, pp. 4-5]. These agents realize greater autonomy and are more adaptive, though they may still be designed for specific tasks.

In practice, there is not one agent type that is always superior. Instead, the different types are suited to solving different problems. Unland notes that some researchers have claimed that reactive agents, particularly used in combination, are just as powerful as deliberative agents, and in practice hybrid solutions, as well as federated "multi-agent systems" are often preferred [14, p. 5].

Replicating many of the advantages seen above in relation to the use of microservices architectures, these systems can achieve sophisticated results using simple building blocks [14, p. 7]. In their most sophisticated form, they may exhibit qualities of highly advanced autonomous systems such as self-organizing, emergent structures and

"swarm intelligence" mimicking insect behavior [14, pp. 14-15]. It is clear then that the possibilities for agents are wide-ranging, and they promise to expand further in the future.

2.3.2 Challenges of Agent Design

To complicate matters somewhat, the range of capacities is not the only concern relevant to agent design or selection. More advanced deliberative agents are undoubtedly more capable, but they introduce more complex challenges as well. If agents are fully autonomous, making decisions in place of human users, their designers must confront the difficulties and ethical implications involved in deciding what guidelines they will abide by in doing so, and the answers may be far from obvious.

In the context of self-driving cars, for example, the connection to the so-called "trolley problem," first described by philosopher Phillipa Foot in 1967, is often cited. In Foot's famous thought experiment, an engineer driving a train must choose between continuing down the same track, endangering five innocent lives, or changing to another track, where one innocent person will certainly be hit [23]. Various versions of the scenario test humans' competing ethical intuitions. It is quite clear that autonomous driving algorithms will inevitably face similar decisions where lives are on the line.

In a more generalized application of the same problem, Shams et al. raise the important question of how autonomous agents pursuing their goals should weigh the various norms they are designed to respect [24, p. 2]. Ranking norms against each other is itself an important and difficult question, but it must also be asked whether and when norms should be violated when doing so better serves the agent's goals [24, p. 2].

While it will become increasingly important for programmers to consider issues such as these as autonomous software becomes more sophisticated, it remains the case that dilemmas of this kind are not faced by the majority of software agents today, as their capabilities are more limited. For the goal of enhancing a data collection agent, it is much more the reactive sort of agent that is of immediate interest. While the ideal DCA may be capable of advanced decision-making, a first approximation in a semi-autonomous agent will be a simpler implementation of rule-following.

2.3.3 The Data Collection Agent

The agent under study in this project, a data collection agent, was conceived and developed as a program suite incorporating a library of modules for the collection of data through various means. The agent runs on a virtual or physical hardware node and sits at the edge of the system, where it may be connected to various sensors or data sources.

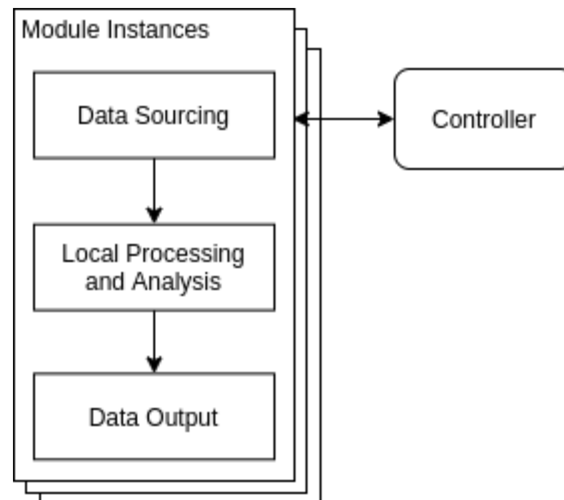

Figure 3. The DCA suite exhibits a modular structure, with running instances managed by a controller process. Modified from confidential internal documentation.

Figure 3 above depicts the controller and modules of the DCA. The controller is the central process that launches appropriate module instances according to the DCA's configuration. Modules may be specifically tailored to a particular use or generic enough that they may be launched simultaneously in multiple configurations.

The figure also illustrates each module's internal three-stage structure. This includes a connection to some type of data source, whether a local simulation, local physical sensor, or external source, such as data from the Internet. The data may then undergo some initial processing or analysis to derive the desired event information, after which it is passed out of the DCA as output on the system's message bus.

As seen here, the structure of the DCA is at the same time relatively simple but susceptible to failure at various points. It presents a number of challenges for further development related to monitoring, restarting, and reconfiguring its various components.

2.4 System Context and Architecture

The system whose development serves as background to this project is an end-to-end platform for collecting and visualizing data about many kind of processes that a user might want to monitor simultaneously. Ranging from sensor output to local weather statistics to data collected by individuals carrying personal devices to information about items moving through a warehouse, the possibilities for what could be monitored using the system are wide-ranging.

The basic motivation for this system's creation was to provide a ready-made solution that could be scaled for different use cases and tailored to a particular organization's needs. While several of the system components are freely available open source projects, creating a comprehensive solution for this kind of data aggregation, analysis, and presentation would be prohibitively difficult for the average non-technical user.

As an MSA-based platform, the system includes microservices for all stages of the data pipeline, from initial collection in the DCA, through the backend build around a central message broker, to the presentation stage with a customized frontend featuring a timeline-based dashboard. All of these basic system components are in a working state, and the system as a whole has been subject to in-house testing over a period of several months. The present project plays a part in a larger goal for the system of developing additional semi-autonomous controls for the various component services. The project of developing a semi-autonomous data collection agent, therefore, will also be applicable to similar improvements throughout the system.

Figure 4 below depicts an example dashboard in the system's user interface. Customizable panels correspond to specific streams of event information, which can be displayed as numeric graphs with dots superimposed to draw attention to events of particular importance or dots alone, depending on the nature of the monitored data. Additional contextual information about events is provided by a custom tooltip.

Figure 4. The system’s user interface displays a variety of information about monitored events.

As the figure underscores, the system’s foundation on an event-driven MSA means that each service is oriented toward creating, managing, or altering streams of events. Building on the principles of MSA design described previously, an event-driven architecture adds some additional concepts that further enhance many MSA strengths. Mikkelsen et al. define an event-driven MSA as one centered around “message-based communication … where interactions are asynchronous and handled by a discrete data object known as an Event” [9, p. 6570].

In other words, in an event-driven MSA, coordination among the various system services is handled not on the basis of synchronous calls but on the premise that data shared among the services belongs to identifiable events that originate at known times. This leads to the concept of application states (Mikkelsen et al. compare these to the running balances of bank accounts) as summaries of events leading up to the present time, allowing for later replay and rebuilding of intermediate states, along with the possibility of injecting or removing new events that alter these states [9, p. 6570].

The present system consists of five fully functional services with plans for additional expansions. While, in theory, any of the services could be duplicated or scaled, the DCA in particular was designed to run in multiple side-by-side configurations as part of a single system. In that sense, it is a single service type that can be run as multiple microservices.

The primary event data flow begins from the point of collection in the DCA and continues to the message broker service, which serves as the backbone of the entire system. The broker, which is based on the open source Apache Kafka project, is capable of running in cluster configurations to ensure the kind of redundancy needed to avoid catastrophic failures, a potential problem discussed in the section on MSA above.

As seen in figure 5 below, data output from the DCA flows through the message broker, and the primary data flow then continues to the data formatter service, which reformats Kafka data into the format accepted by the long-term storage solution, implemented using an InfluxDB database server. An expansion in development will add the capability for live stream processing prior to long-term storage, which will also be used as the basis for the planned implementation of alerting and other trigger-based actions. After entry into an InfluxDB database, the data visualization is handled in the last major service currently implemented, the user interface service. The Web user interface is based on the Grafana project with the addition of a custom panel plugin developed to meet the system's specific presentation requirements.

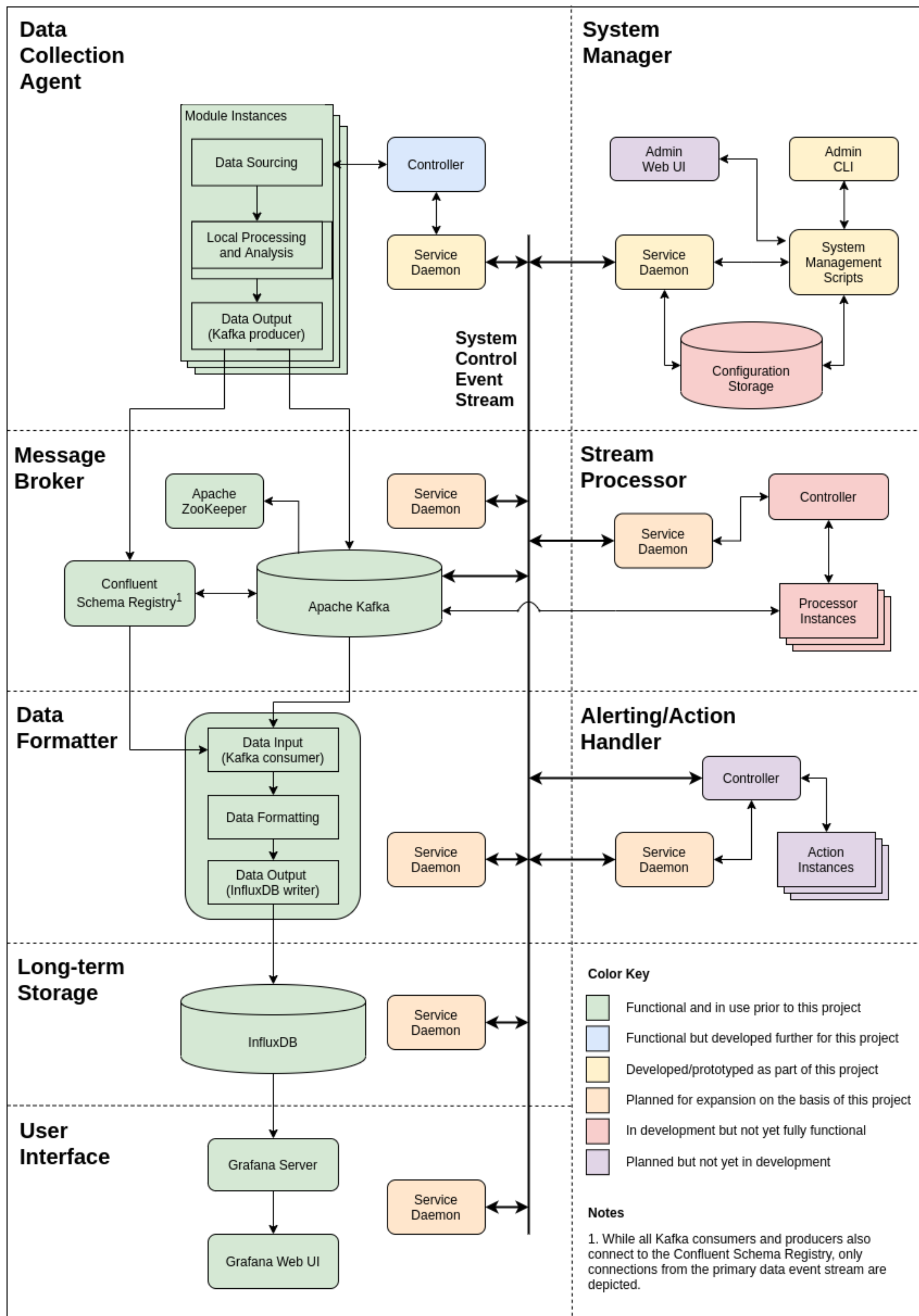

Figure 5. The system architecture is based on an event-driven MSA. Modified from confidential internal documentation.

In addition to the parts of the system that have already been in internal testing for many months, figure 5 also depicts planned expansions to the system. These additional services, which will provide features such as user-customizable alerting, will contribute to the larger goals of making interactions with the system easier for users, reducing the need for users to actively monitor the system, and requiring less intervention from administrators. All of these goals are also served by the enhancements to the DCA in this project.

2.5 Intended Contributions of this Project

All of the background in the preceding sections illustrates the reasoning for the existing structure of the system, the role of the DCA in that system, and the motivation for adding degrees of autonomous control to the DCA. As a microservice in an event-driven system, for example, the DCA should strive toward the goal of independent operation and self-maintenance as much as possible. Ideally, it should be able to both monitor its own modules and heal itself in the case of failures.

The theoretical information on automation also helps to focus this project. The type of automation that is most desirable and attainable for the current DCA is a species of what King calls unattended automation and a realization of Unland’s reactive agent type. This project aims to create an enhanced DCA that can react on its own to events it observes as well as messages from other services according to a set of action rules.

Ultimately, this project seeks to implement this reactivity not only in terms of self-healing capabilities but also in self-adaptive behaviors. By creating a DCA that is able to reconfigure itself, drawing from its library of modules, this project will explore the application of different autonomous behaviors and contribute something useful that advances the larger system and has broader application throughout its different services.

3 Methods and Implementation

The project of creating a DCA with self-healing and self-adaptive behaviors began in a collaborative process of planning. By discussing the different capabilities of the DCA, its role in the system, and how it could be made to work more smoothly, it was possible to translate the general project goals into specific tasks. After selecting appropriate tools and preparing a test environment, it was then possible to begin prototyping.

3.1 Defining Implementation Aims

Going into the project, the principle advantage of working with the DCA was its existing modular structure. Having already implemented these functional modules along with a basic controller that could restart them and log any errors, there was a clear path forward for giving the DCA more self-awareness of its various states. This also meant that the DCA was already quite easy to reconfigure, although this could be improved by removing the need to manually update configurations.

At the initial stage, the practical aims of the project for implementing the desired degree of autonomous control were focused around four points. It was determined that the most advantageous feature additions for a semi-autonomous DCA would be the following:

- increased self-monitoring capabilities, with an event stream tracking the status of the software modules and the hardware on which they run
- an interface for the DCA on a system control channel, through which it could receive configuration changes and send requests for needed action
- mechanisms for the DCA to act on the information it monitors, including the ability to restart on failure
- use of local stream processing—or basic analysis of the data collected—to further enhance the DCA's self-monitoring capabilities.

All of these features would contribute to the goals of allowing the DCA to do more on its own to correct issues and providing more information and insight about the system's status to the user. They would also enable future expansion of the system. While it was outside the scope of the current project, this kind of telemetry information could be utilized by the planned stream processor and alerting/action handler to add an additional layer of autonomy to the system. More complex analysis of the performance of all system

components and services, combined with customizable alerting, would take the goal of an easy-to-use, self-managing system to a new level.

What also became apparent at this early stage, however, was that the planned additions to the DCA were quite narrowly focused. Through discussion with the team, it was decided that these features would be even more beneficial if they could be implemented in a way that could be reused for other system services. That focus on reusability as key to automating system functions became an additional important goal of this project and resulted in some implementation details moving toward more generic solutions.

3.2 Tools and Technologies

As with any work on a new component or feature in a larger system, some of the choices of tools for use in this project were influenced or determined by choices made during earlier work on other parts of the system. While this imposed a kind of constraint on the path this project's execution would follow, it also made it easier to get started by clarifying some of the requirements.

These effects of previous decisions also demonstrated the benefit of early planning. Many of the same considerations that would go into design choices for this project were already clear at the time of creation of the earliest system prototype. This was particularly true in the choice of programming languages for various system components.

3.2.1 Programming Language

When the original DCA was planned, the goals of implementing a modular design and allowing for future expandability were central priorities. The program also needed to run in a Linux environment, either on server hardware or in a virtualized container, with relatively few dependencies, to ensure portability. Finally, it had to cooperate with the other technologies selected for the system, such as Apache Kafka, whose producer and consumer functionalities could be easily used with existing libraries for certain languages.

All of these considerations led to the selection of Python as the logical choice for DCA development. In addition to its familiarity and ease of use, Python allowed for the creation of a suite of modules that could easily be imported in other files, contributing toward the

goal of reusing existing work. Python also offered a relatively simple approach to implementing multithreaded programs that fit well with the concept of a controller program running a configurable set of modules. The language also provided compatibility with all other existing system services and components with a minimum of additional work. It is relatively ubiquitous, present by default in most Linux systems, and it allowed for easy installation of needed libraries.

This choice then carried over to the present project. Here, again, the ease of importing new classes and modules from external files, and the ease of creating and managing threads in Python programs were essential for achieving the project's goals. As additional benefits, the fact of Python's being an interpreted language and the ease of testing features in IDLE, its included development environment, allowed for an increased pace of feature iteration.

3.2.2 Configuration File Format

An additional choice that needed to be made for this project was a format for the small text files storing the various configurations and states of the service components and modules. As with the programming environment, the choice of formats was initially made prior to this project's inception, and it was suitable for this project to preserve that choice.

The YAML data serialization format was chosen for these files for a few reasons. First, it was a suitable choice for use with Python programs because of the availability of libraries for interpreting entries in the YAML format as Python objects. It allows for configuration hierarchies and structures that translate directly as Python dictionaries, for example.

```yaml
name: YAML Configuration Example
version: 1.0
embedded_dict:
  key1: value1
  key2: value2
  key3: value3
embedded_list:
  - name: entry1
    type: list_entry
    version: 0.0.1
  - name: entry2
    type: list_entry
    version: 0.0.1
project: Example Project
filepath: "/opt/examples/example.yaml"
```

Figure 6. An example YAML file. The YAML file format is convenient for programmatic use while still legible to and editable by human users.

Figure 6 above depicts an example of a YAML-formatted configuration file. In this example, the embedded dictionary and second-level list are readily translated in Python code as a dictionary and a dictionary of dictionaries, respectively. This example also illustrates a second major benefit of the YAML format: it is useful not only for automated processes but also for humans. It provides a legible, plain-text format that human users can edit manually. This made testing different configurations much simpler and contributed to the system-wide goal of preserving some degree of manual user-configurability even while moving toward the incorporation of semi-autonomous functionality.

3.2.3 Service Process Management

The choice of mechanisms for starting and stopping the processes of the system's microservices was another key decision that would help shape the project. The decision to use systemd for this purpose was, as with other choices of tools, largely influenced by previous decisions made during the course of developing the system. Some of the technologies adopted for the system, such as Apache Kafka and the Confluent Schema Registry already used provided systemd service files to start and stop their processes and to provide a method for enabling them to run automatically at machine startup.

```
[Unit]
Requires=network.target remote-fs.target
After=network.target remote-fs.target

[Service]
Type=simple
User=exampleuser
WorkingDirectory=/home/exampleuser
ExecStart=/home/exampleuser/example.py
Restart=on-abnormal

[Install]
WantedBy=multi-user.target
```

Figure 7. This example systemd service will execute a Python script when started.

As illustrated in figure 7 above, systemd service files are quite simple but define a variety of parameters for the execution of programs, including the working directory, restarting behavior, and any dependency relationships with other services. This feature allows users to create multiple services that start and stop in a defined order, something that would prove important for this project.

A relatively recent addition to many Linux distributions whose adoption was widely seen as controversial, systemd is described in the Ubuntu Linux manual pages as “a system and service manager” that “brings up and maintains userspace services” [25]. It defines a variety of types of so-called “units,” with the most important of these being the “service units” that are commonly used to manage the running of processes [25]. A simple shell command can initiate service processes or cause them to exit cleanly.

With the provided facility for user-defined units, this makes systemd an ideal method of handling processes for microservices that do have dependencies and need to be started, stopped, and restarted in a controlled manner. Through its associated commands, such as systemctl and journalctl, systemd also provides methods of checking the status of its services—whether they are running, are inactive, or have failed for some reason—and viewing logs including the standard output and error printing of associated processes. These features would be important later in the course of this project.

3.2.4 Testing Setup

A final important tool for use in the development of this project was the testing setup. This, too, was determined as a result of both the other tools chosen for the project as well as the preceding work. This meant, in practice, that development and testing was done primarily on local hardware running Linux and Python.

The key benefit for testing changes to the DCA service, however, came from the benefit of a running prototype system installation, with all of the core system services functional. With the system having been set up much earlier, running successfully for a number of months, it provided a relatively stable environment in which new features and changes could be tested in a realistic use case, seeing their effects throughout the other services. As this project aimed to improve the existing system, this testing of new functionality incorporated into the system was essential.

For testing the semi-autonomous DCA with the real system, new files were copied to virtual machines running on system servers and were then tested both by running them directly in an interactive shell and by reconfiguring and restarting a running DCA. In the case of testing with DCA modules collecting physical sensor data, a Raspberry-Pi-based device was also used, running the same code.

3.3 Choosing an Implementation Strategy

As detailed earlier, the DCA service as it existed prior to this project consisted of a controller program that could create and manage threads running instances of data sourcing modules. These modules, all written according to the same template, were created to be run in various combinations, all managed by the controller. Many DCAs with different configurations could also be run simultaneously.

The features of the controller offered a compelling starting point from which to build more autonomous control of the service. A key advantage provided by the DCA controller at the outset of the project was that it had been planned with some functionality for self-management. Because the DCA had been conceived originally with this modular design, it was important that its controller be able not only to start module instance threads but also restart them in the case of recurrent data collection errors or fatal errors that would end a thread's operation entirely.

The DCA controller accomplishes this task through the common design of the modules, by creating and managing threads for configured modules in the same way: initializing a data collection class object, starting its data production loop—in which it repeatedly collects new data and produces to the message bus, either according to a configured interval or on sensor interrupt—and monitoring the status code returned by the producer for errors. In the case of errors where data collection is interrupted or prevented, the controller's thread management reinitializes a new object and restarts the loop. These events are also logged, allowing administrators to see the history of their occurrence, along with contextual information about the errors.

While this was quite useful already, it was important to consider whether building on these features limited the direction and application of the project too much. Expansion and improvement of these thread management features was a compelling option for further development. At the same time, the controller did not have the capability to make any complex judgments about when to restart threads, had no capacity to fix problems beyond mere restarting—which might frequently result only in the same erroneous behavior, depending on its cause—and its restarting behavior lacked any direct configurability, so it could not be changed without modifying code.

Most importantly, focusing on improvements to the DCA controller would limit the gains of the project for the larger system. As mentioned previously, the team had made an important strategy decision at the outset of the project that improvements to the DCA's self-awareness and self-control should not be strictly limited to that service. While the DCA could still be a focal point for development and testing, this pointed toward a more generically applicable implementation.

This better alternative was found in the creation of a "service daemon," a program always running silently alongside a service, monitoring and managing it. The daemon could be used not only with the DCA but in combination with any of the system's existing or planned microservices. It would also serve as an important link between these services. This meant that the project remained focused on semi-autonomous improvements to the DCA, but it would achieve these results in a different way than had been originally envisaged. Each subsequent decision would therefore be made with the broader application and implications of the project in mind.

3.4 Conceptualizing the Service Daemon

With the goal of reusability in mind, the process of designing and prototyping a generic service daemon became the most important task of this project. The daemon was principally designed for and tested with the DCA as its assigned service, and its design wound up incorporating a component based on the existing DCA service. Yet, at the same time, it built toward a result that would improve every system service and bring greater automation to the system as a whole rather than focusing narrowly on the DCA.

With the intention that it would become a cooperative entity that would manage the semi-autonomous DCA, as well as other services, the service daemon was conceived from the start as a program consisting of multiple threads. Each thread would handle some specific task associated with managing the service. The daemon needed threads for each of the following planned tasks or roles: monitoring service operations and the health of running processes, monitoring local hardware resources, handling communications, and acting on the information gained through monitoring. It was clear that these tasks would be handled best by separate threads as they would necessarily require simultaneous operation. In fact, the complexity of these tasks would require even those planned threads to spawn additional threads.

At the outset, however, work had to stay focused on just those tasks that were deemed most essential. It was decided that the best method of designing the daemon would be to create successive prototypes with new features. The first version would stay as simple as possible to create a working program. After that, new prototypes would progress toward more complex behaviors. This would also resemble the model of rolling updates to microservices, in which new improvements can be continuously applied to components of existing systems without any major disruption to operations.

In addition to these considerations, the fact that the daemon itself would be added as a separately running process—indeed, one with many threads—meant that it would also need some mechanism for monitoring its own operational states. In other words, it was clear that for each new process added, and each thread running within that process, the tasks of keeping the service running and knowing its states would grow more complicated as well. The new monitor would also need to be monitored, adding some complexity to the task of designing the daemon.

During the initial planning phase, it was also important to consider whether the service daemon's code should be customized for its assigned service or whether the same daemon should run everywhere throughout the system. This was particularly important in relation to the planned system manager service (depicted in the system diagram included as figure 5 above) that would exercise more control over other services and therefore need additional capabilities, but the question was relevant even in the case of the DCA, whose specific operations are unlike most other services.

Adhering to the goal of reusing work, it was ultimately decided that one daemon should be used throughout the system, with its capabilities being set but their use being configurable for the particular service it was assigned to. This simplified the development process, yet it also highlighted an important consideration for testing: the prototype daemons would need to be able to take on different roles such as "DCA service daemon" and "system manager daemon" in order to test the state of features such as inter-service communications and complex triggered actions such as updating configurations.

3.5 Communication Roles and Interfaces

The most essential threads needed for the service daemon were those related to system communications. While a single communications thread had been planned, it quickly became apparent that multiple interfaces were needed, and the number of threads would need to increase accordingly. In combination, these threads would provide paths for communications among processes, services, and human administrators.

First, the service daemon needed to be able to talk to other system services. The most straightforward approach, using existing components to reduce unnecessary duplication of features, was to implement a system control signal as a new data stream carried by the message bus service. In other words, a new Apache Kafka topic would be created, in which specially formatted messages would be sent and received by, for example, the DCA and system manager daemons. This would allow not only for requests and commands to be sent between these services but for more complex behaviors, such as synchronization of system tasks distributed across multiple services.

As the creation of new Kafka topics is as simple as producing new messages to those topics, the creation of the system control signal was as simple as importing Kafka producer and consumer Python classes in the daemon thread. Messages published to this topic can be addressed, so that each daemon needs only to check for its own address to verify that it is the intended recipient of a given message. As all daemons are intended to share the same signal, this also allows for the possibility of group messages.

Two additional considerations contributed to the design of the control signal implementation. As producing and consuming Kafka messages constituted two different roles, which would use different class objects and potentially need to operate simultaneously, the most logical approach was to create an additional thread. This meant that the main control signal thread would become the message consumer and would create a secondary thread for message production.

Additionally, other threads in the daemon process would need to be able to trigger production of messages on the control signal in response to various events. To achieve this, a Python queue was created for messages pending production to Kafka. Rather than having its production function called directly, then, the producer thread would poll the producing queue for new messages with a timeout in the case of an empty queue. New messages could be written from any thread into the shared queue.

At the same time, a second communications channel was needed. Because the decision had been made to implement the service daemon as an independent process running separately from its assigned service, such as the DCA, it was decided that it would need the capability of communicating with other local processes, within the scope of the single service and on the same machine. This channel could then also serve a second purpose as the planned interface for administrative scripts that needed to communicate with the system manager service's daemon, ensuring administrators would maintain direct control over the DCA and other services.

For the creation of this interface, the Python multiprocessing library's Listener and Client classes were selected for their ease of implementation and suitability for the intended purpose. The listener-client model would allow multiple simultaneous connections if needed. This was implemented by creating a main listener thread waiting for connections through sockets on the local host machine. As each new connection is accepted, a new thread is spawned to handle the exchanged communications for the lifetime of that inter-

process connection. Figure 8 below depicts the concept for the daemon's communications flow over both interfaces.

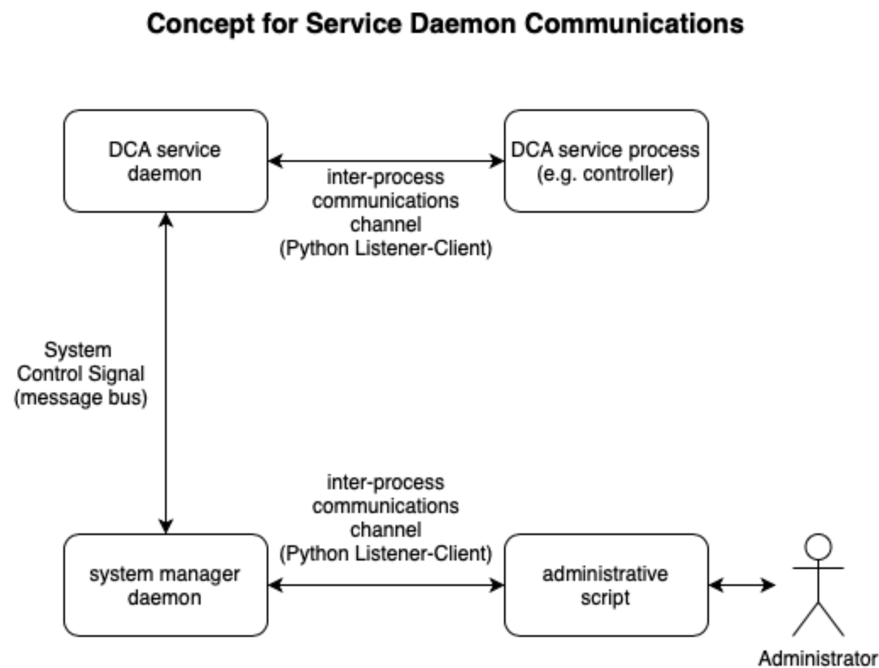

Figure 8. The communications interfaces were conceived to connect administrators to services.

The testing of communications in the first daemon prototypes was initially relatively straightforward. With regard to the system control signal, this required running two instances of the daemon, which could then exchange messages using addressing. At first, these messages were designed to simply reply automatically to each other, with no meaningful content analysis or follow-up action, to verify that the production and consumption of messages worked as expected.

Testing of the inter-process listener-client communications channel was more complicated, as it required the creation of a client program to talk to the service daemon. The first iteration of this client simply passed through user-entered messages from a terminal, sending them directly to the daemon prototype, which would then acknowledge the messages with a set response. This client served as a stand-in for service processes or the planned suite of administrative scripts that would be able to talk to the service daemon.

The need for more complex analysis of received messages, with more variable response types and triggered actions, pointed toward the creation of a more sophisticated daemon

configuration, which would incorporate rules for handling predefined messages, something that would be developed in later prototypes (see section 3.6 below). Inter-process communications with the client program would also be developed further to work cooperatively with the control signal to transmit files between daemon instances. This would achieve the additional goal of implementing a mechanism for updating configurations of the service automatically.

A final important addition to the communications capabilities of the daemon was the addition of the basic functionality for more advances request handling. While the daemon was already capable of responding to incoming messages (and the addition of configured responses would allow it to handle any requests and associated actions that the user wished to configure), this would not guarantee that any triggered actions would indeed execute successfully or that meaningful responses could be sent. The daemon or the administrator intiating the request for action would need information about the outcome of the request, whether the action succeeded or failed, and what the receiving daemon did in the event of a failure.

This necessitated the creation of some persistent request handling mechanism, a task that would be spawned in response to the request and kept alive until the request had been fulfilled. Analysis of this problem suggested a few possibilities for implementing the handling functionality. Incoming requests could cause new threads to be created that would run alongside the primary service daemon threads, communicating with them and delivering the eventual response. Handling could also be implemented using a queue that would contain information about received requests, the needed actions, and the possible replies that could sent. These solutions were both problematic, however.

The key challenge for implementing request handling was that the handling mechanism needed to be able to outlive the daemon process itself. If a service daemon were to exit prematurely, due to some system software error or hardware failure, or if it simply needed to restart, any threads it created would be killed. In addition to this, the contents of a queue would not persist beyond the lifetime of the process, unless they were backed up in some fashion.

This led to a different solution to the problem of request handling, based on the use of an external handler file. The file, in YAML format and containing an ordered list of entries,

would serve as the queue of received requests that needed to be handled. As each incoming request was written to the file, the daemon would also read the file for any existing requests, and it would respond to them according to the information stored, until all requests pending at that time were handled. This process is illustrated in figure 9 below.

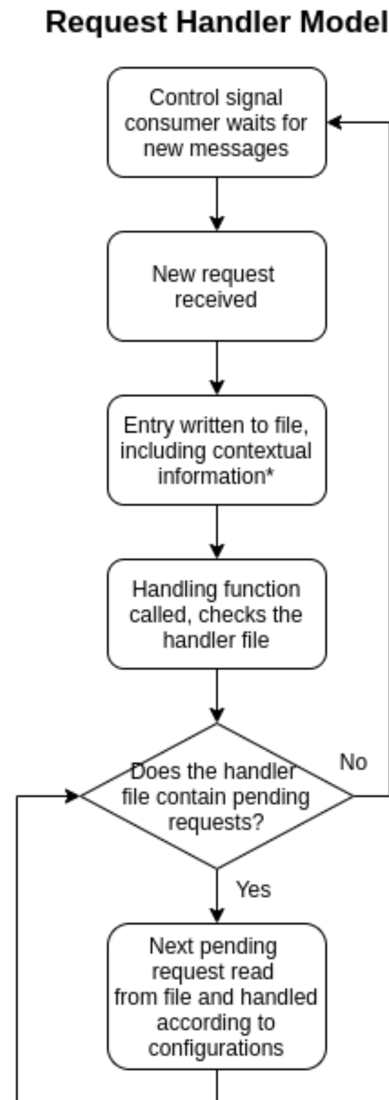

Figure 9. This model for request handling was chosen for the system control signal.

One consideration raised by this solution was that it could potentially cause a delay in request handling upon a daemon restart, as handling of pending requests from the file would be triggered only upon receiving a new request. The alternative of a timed, periodic checker that would read the handler file was considered, but this was ultimately deemed unnecessary, as requests are intended to be infrequent events, and they should be handled relatively quickly, meaning that it would be unlikely that numerous entries would

ever remain pending in the file. In addition to this, daemon restarts would ideally be infrequent, meaning that most requests would be handled right away.

While in theory this approach could also block receipt of new messages, this too was not necessarily problematic. Messages requiring more complex handling would be received over the system control signal, whereas inter-process messages would be more direct commands. In this respect, the control signal would benefit from the indexed and buffered nature of Apache Kafka topics, which would ensure that even in the case of a longer delay before resuming consumption, a topic consumer would get the next waiting message. With two threads handling control signal communications, it would also be possible to produce responses and consume requests simultaneously and independently.

3.6 The Service Daemon Configuration File

The solutions implemented for communications already pointed toward the need for complex configurations that would guide the service daemon's behavior, and the implementation of configurations was begun next. As a reactive agent, the daemon would respond to incoming events, including requests received via messages. This meant that a system of guidelines for responding to and action on these requests was essential to the daemon's basic functionality.

The daemon configuration file, read at the start of execution of the main service daemon process, was to contain not only the basic settings needed to establish communications on the daemon's two interfaces but also a full list of defined, anticipated events and what possible actions or responses those events would trigger. These were written in the form of trigger and action rules.

The task of defining rules for linking observed events to resulting actions required thinking through a few different possibilities and answering the following questions:

- What kinds of events would the service daemon potentially encounter?
- What did the daemon need to note when monitoring the condition of the service and local resources?
- What did the daemon need to do in these different circumstances?

These and similar questions had to be answered by the list of rules. It was clear that the configured rule set needed to define sequences: an expected input (received message

contents or monitored program states), a triggered condition met by the input, and a result from that trigger, including a response through one of the communication channels, an action to be taken, or both. The simplest approach to this was to implement a separate list for each service daemon thread, where each would follow its own configuration in order to organize chains of activity to accomplish tasks together with other threads.

The rules were designed to work essentially as a system of related look-up tables of key-value pairs. These were designed to meet the needs of each interface and each thread. For the inter-process communications interface, for example, it was decided that the rules would consist, at least in the first draft, simply of a list of possible message contents, or commands, that might be received from client programs along with assigned trigger codes for each. The triggers would then be entered into a queue for the daemon's action handling thread to process (see section 3.8 below).

For the system control signal, the rules would connect incoming messages to their defined responses, which could include multiple possibilities. They would also include the same action triggers. In the case of a triggered action, the handler would, when fully configured, follow up by monitoring the action's outcome and passing the information about its success or failure back to the sender of the request, using the preconfigured responses.

For monitoring of program states (detailed more fully in section 3.7 below), the rule list would be somewhat more complex. As the idea was to keep some running list of states of various modules or components of the service, the rules needed to define conditions in which these state variables would trigger some response. These conditions might apply either to one particular monitored state variable (e.g. the state of a specific DCA module) or to any of the monitored state variables that were in a given state (e.g. a "critical" or "failed" state). Figure 10 below is an example state diagram depicting various operational states that a DCA module could be in during the program's execution.

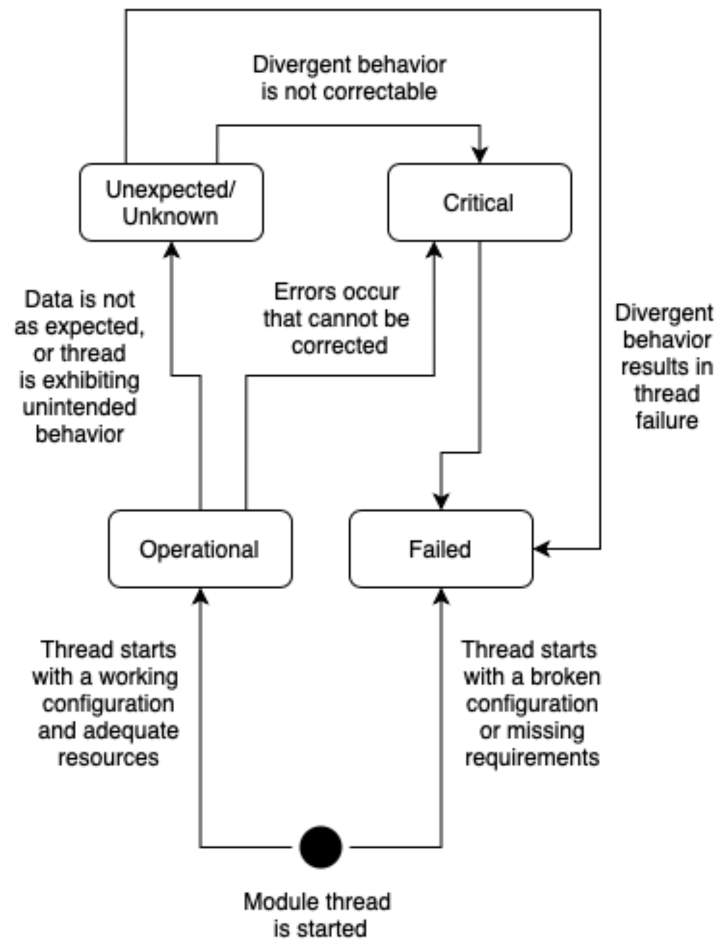

Figure 10. A DCA module thread might be in different operational states at different times.

This was then expanded with some additional considerations that would contribute to a more sophisticated state monitoring capability. It was decided that the state monitoring mechanism might not only need to monitor the current state of a given variable but also how long it had been in that state and how often it was changing state. These data points were therefore included as separate fields in the list of state monitor rules.

Finally, it was anticipated that the state conditions which were of interest for responsive actions might involve combinations of these more basic states of specific variables. For example, what if some response were required in the case where module X had been in the “critical” state for several operational cycles, but only when module Y was also in a “failed” state? Each of these sub-conditions could already be written separately, so a simple mechanism for tying them together would be preferable to a more complicated, and redundant, way of writing them as a single condition. This pointed to a need for group conditions, and these were then implemented as well.

For group condition rules, combined triggers were created alongside the trigger identification numbers used by the simple conditions. Whereas a simple condition, when met, might result in trigger “101” being added to the trigger queue, for example, the associated trigger for a sub-condition which was one part of a combined group condition would be written as an array of two integers—the group condition identifier and a member number for that sub-condition.

A separate list of group rules would then keep track of the number of member conditions in a group and the secondary trigger to be added to the queue once all conditions in the group were triggered. Figure 11 below illustrates the process of checking both simple conditions and combined group conditions and issuing the associated triggers in a simplified form (in the real implementation, the action handling thread is also involved in the process of tracking which group member conditions have been met, and it saves this information, which is then checked by the state monitor thread).

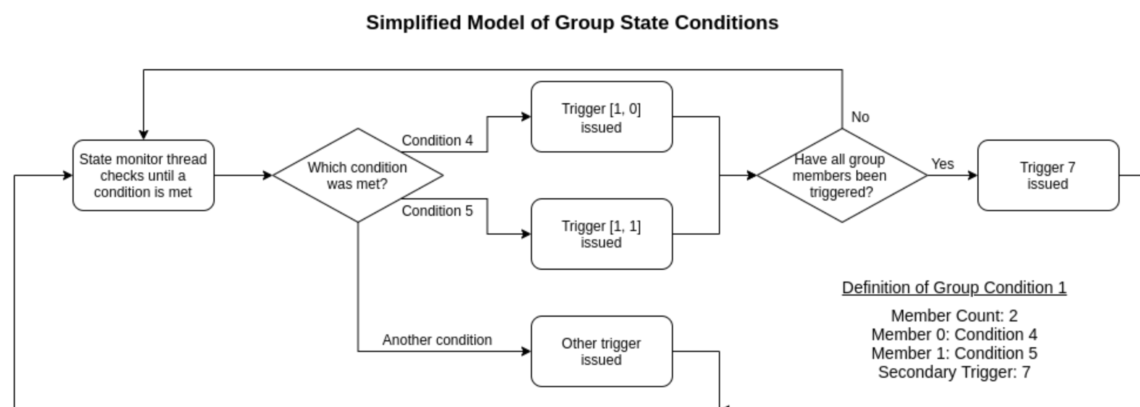

Figure 11. Group conditions are defined as combinations of multiple simple conditions.

The final category of rules that had to be written were for the daemon’s action handling thread. Action handler rules would map all defined action triggers, those connected to message contents in the other rule lists, to actions. The actions could be of various types—shell commands to be executed automatically, log statements to be printed, and so forth. This left open the possibility of relatively simple expansion with new ideas in the future.

A final type of actions were those for group conditions, as mentioned above. Group actions would set flags to keep track of the triggered member sub-conditions. The secondary triggers associated with groups whose conditions were all met would then be associated with other action types.

Metropolia University of Applied Sciences

All told, these different sets of rule lists, with different purposes and providing different information, created a working chain of command, where conditions met in any of the several service daemon threads would be carried through to result in the configured actions. The mechanism created would allow for conditions as simple or as complicated as the user might desire to create.

3.7 State Monitoring and the Integrated DCA

While the monitoring of the various states of the service had been taken into account in the creation of rules and conditions, the next feature that needed to be implemented for the project was the thread that would actually keep track of these states, as well as some kind of data store where they could be maintained in a way that would persist even in the case of a restarted process. As an additional tool for monitoring these states, an integrated version of the DCA service (or iDCA) was conceived, which would become an important part of the generic service daemon.

It became clear right away, when beginning the implementation of this feature, that a variety of different states needed to be monitored. These included the states of the DCA service, or other service associated with the daemon, the daemon process itself, including all of its threads, and the condition of local hardware resources. The clearest solution to this challenge was to separate these different states and to store them in separate files. Figure 12 below illustrates the model for implementing these state stores and depicts the various threads of the service daemon and iDCA.

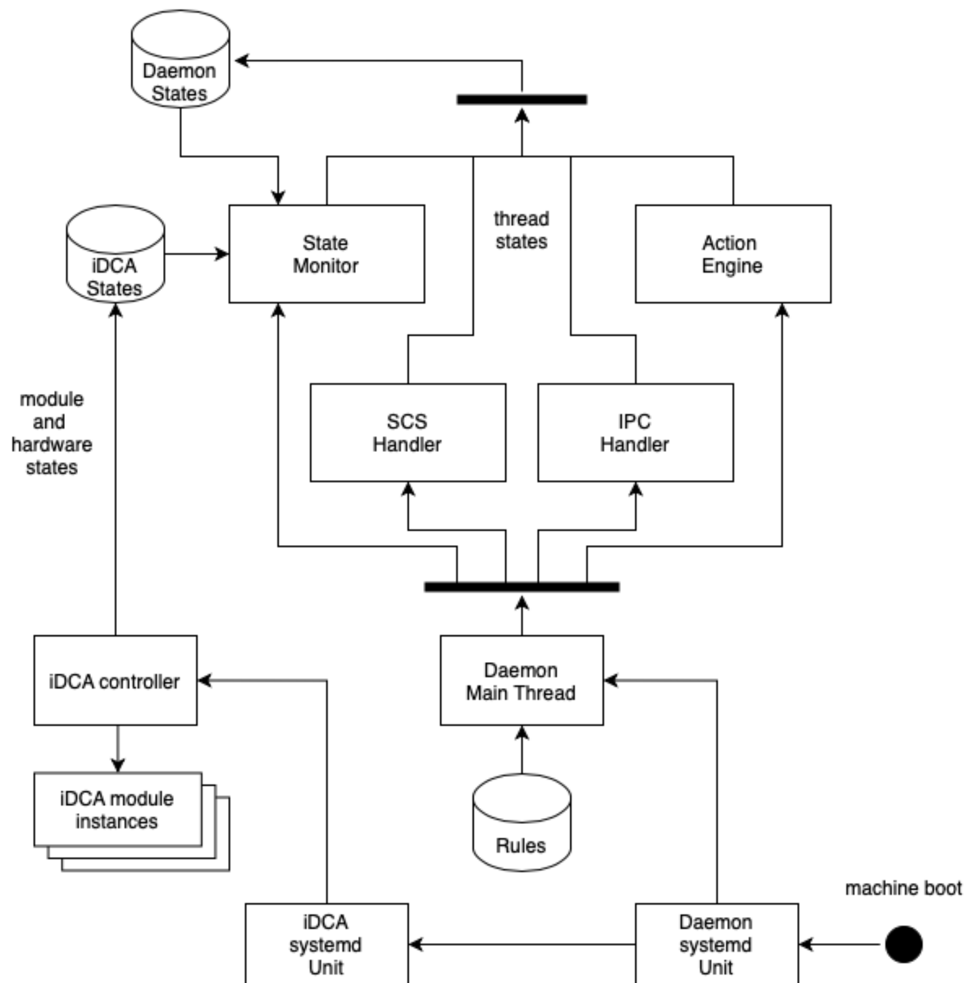

Figure 12. The threads and modules of the daemon and integrated DCA and their state stores are modelled. Based on confidential internal communications.

Illustrated on the left side of figure 12, the first state variables to be monitored in a service daemon prototype were those related to local machine resource measurements. These kinds of measurements—such as CPU, RAM, and disk space usage—had in fact previously been tracked during the development of the system in the form of DCA modules, so monitoring them as state variables was relatively straightforward.

These existing modules provided the basis for what would become the iDCA, a specialized version of the DCA service that would run only the specific set of modules designed for monitoring of local resource usage and machine metrics. Each of these quantities monitored by the iDCA would translate to a state variable that would be saved to a file, and that file would then be read continuously by the daemon's state monitor thread,

which would act on these measurements according to the configured rules (e.g. triggering the restarting of a process, or a local machine, or reducing the frequency of certain operations on the basis of critical resource levels).

It was ultimately decided that the monitoring of the status of the DCA service itself would also be implemented as a module for the iDCA. Although the states of both DCAs could have been monitored in the same way, it was decided that the regular DCA service, as one service among many, should be monitored in a way that would apply to services more broadly (most of which were not susceptible to the same time of monitoring implemented for iDCA threads), whereas each service daemon would be accompanied by its own iDCA. The module developed for monitoring the status of the DCA service is discussed in section 3.10 below.

For the monitoring of the states of the service daemon process itself, a separate file was created, in which the daemon would record the active or inactive status of each of the threads created when it runs. After the point of initialization, where the configurations are imported, various files opened for reading and writing, and the different threads created and started, the main thread of the service daemon was repurposed as a state updating thread.

After this point, the main thread would record the status of each of the threads it had created during the remaining time of normal execution of the daemon program. This process would continue indefinitely, until the setting of the daemon's stop event—a Python threading library event, created to ensure the smooth exiting of all running daemon threads and linked to the system interrupt and termination signals.

One last additional file that would be added for the tracking states of the service daemon, and an addition to the group state condition checking mechanism, was the group state file. In this file, the status of each configured group for which any member sub-condition had been triggered would be recorded during the daemon's execution. This ensured the persistence of the states of the multi-condition groups, so that the associated triggered actions would be executed as intended even after restarting the service daemon.

3.8 Action Handling

The final main thread that needed to be implemented for the service daemon’s basic functionality requirements to be met was a thread for initiating action based on received requests or events seen during the operation of the daemon’s self-monitoring facilities. As mentioned previously, the plan had been to support a variety of action types, and mechanisms for each action needed to be implemented in this thread.

The action handler would operate on the basis of the trigger queue and configured rules discussed in earlier sections. Removing items from the queue one by one, the thread was designed to look up the associated action type and action contents for each new trigger. The thread would then use conditional logic based on action types to determine what tasks to execute.

As a benefit of running the process from systemd, the service daemon could safely run as the root user, meaning that the action handler could execute privileged commands without the need for manual authorization from an administrator. Allowing root access meant that securing access to the Python source files on system machines was even more crucial, but this level of security had already been established as a background assumption. Root access also inherently introduces the possibility of highly destructive bugs, but the benefits it provides were critical to allow the program to act independently.

This made the action handler capable of executing tasks such as starting and stopping systemd services, accessing log entries for those services, restarting the machine on which the daemon is running, and other similar tasks. This again pointed to the importance of configurations as the key to enabling the kinds of actions that would be most beneficial if done independently by the service daemon.

3.9 Configuration Updates

One additional specialized action type, implemented as part of the action handler but of particular importance to the overall project goal of increased autonomy, was a mechanism for updating configurations. This could include configurations of the DCA service, the service daemon’s iDCA, or even the service daemon itself.

Separate from the present project, work had been undertaken on the creation of an administrative script for generating configurations for the system's microservices. Once new versions of these configurations were generated, however, the question remained of how to get them to each service and to cause the services to begin using them. This had been done manually, but the goal was to automate this process as well.

To allow the system manager daemon to send the new configuration files to the machine where the DCA service daemon was running, multiple possible solutions were considered. One would be copying via scp, which the daemons could do in theory because of their privileged access. Yet a more compelling option was to use the new communications interfaces. As messages in Apache Kafka could be large enough to contain the contents of entire files in a field, along with important contextual information about which file was being sent, the decision was made to adopt that method for sending files as attachments, addressed to the target service daemon.

A configuration action type was then also created, along with a separate Python script for updating the configurations and testing them. A testing mechanism with the possibility of rolling back to a previous working configuration was crucial, because wrongly formatted, incomplete, or broken configurations could take down the DCA service and therefore break the system.

After these additions, when a new configuration was received via the system control signal as an attachment, the triggered action would save the received file contents under a received name according to a defined folder structure for archiving configurations. The associated action script would then determine which kind of configuration had been received, would replace the active configuration with the new one, and would then issue the command to restart the relevant service process—the DCA, the iDCA, or the service daemon.

To enable the possibility of following up after a configuration change to ensure the successful operation of the service, the updating script needed to be able to outlive the main daemon process, in the case where its own configuration was updated. For this purpose, then, a new thread was created for the updating script using a double fork, to spawn a process that would not be killed upon the exit of its original parent process.

In the end, then, the updating process would continue execution until the reconfigured process either resumed successful operation or failed. Upon a failure, it would restore the last known working version of the relevant configuration and again issue a restart command. This would ensure that, at the end of the update attempt, all constituent processes of the service were operating normally.

3.10 Monitoring DCA Service Status

For semi-autonomous control of the DCA service, the most important function that had not yet been implemented in the initial service daemon prototypes was a mechanism for monitoring the status of that service. This had initially been a primary goal of the project, yet it was saved for later implementation for two reasons: the shift to a focus on more generic solutions for service monitoring using the service daemon and the desire to build on the daemon's other functions for the purpose of service monitoring.

Importantly, the decision was made that the DCA service's own states should not be tracked in the same way that iDCA states were, although the two were versions of the same program, with the same modular structure. This was done to preserve the idea that the same service daemon, with the associated iDCA, could run alongside any system service in the same way, not being overly specialized for a particular service. Instead, the specialized mechanism for monitoring service status would be handled as an iDCA service status module, such that similar service status modules could later be developed for all other system services.

To implement the service status module for the DCA, the DCA controller itself first needed to be modified to write the states of its own threads to a file that could then be read by the status module in the iDCA. Because the DCA service, unlike the iDCA, could be running different combinations of modules at different times, with a variable number of running threads, the decision was made to track state-change events rather than tracking a running state on a thread-by-thread basis. While this would sacrifice detail, it would allow the DCA controller to retain responsibility for running module threads, whereas the daemon, through the iDCA, would monitor the service as a whole.

When the DCA controller printed events to the systemd journal, it would now also write one of a new set of defined event codes (e.g. "INIT," "IMPORT," "THREAD_START,"

“ERROR,” etc.) to its own state log file. Only a fixed number of these recorded codes would be kept at any given time, so that the effect was that the file would serve as a moving window of the most recent types of status-change events from the service.

On the other side, the new DCA service status module running in the iDCA would regularly read this file, observe the number and frequency of certain event types, such as errors, and then determine when the DCA service was either operating normally or was in a failed state, triggering either a restart of the service or some other configured action. The criteria for what constituted failures or other important states of the service would benefit from further iterative development and configuration.

3.11 Service Installation and Initialization

While work on this project had progressed to the state that the DCA service could be updated, monitored, and managed by its service daemon, one task that still required an entirely manual approach was initial installation of the service. A final addition to the project that took the work one step further toward semi-autonomous services, therefore, was the addition of a script for initial service installation that was first developed specifically for the DCA service.

This script would complete the semi-autonomous DCA by allowing its files to be copied to the assigned system node by the system manager daemon, which would then start the DCA’s daemon, responsible for running the service. Additionally, as this script would be run by the system manager daemon itself, checking for new installation targets based on a master configuration, this meant that new DCA instances would be able to be launched as needed without the need for step-by-step interaction by system administrators, who would need only to have written or generated the corresponding system configuration files that specified where DCAs should be installed, which modules they should run, and what they should do with the information they collect.

4 Results

Upon completing the prototyping of the various features described in the preceding sections, the development of the service daemon for the DCA had progressed to a state where the foundations of a semi-autonomous agent, which had been the ultimate goal of this project, were established. From service installation to ordinary operation, the entire process of running the DCA had been enhanced through the addition of reactive automated functionality.

The service daemon is now able to effectively install and run instances of the DCA service, starting and stopping them, updating their configurations, and handling communications related to their status. The use of all these procedures together as part of one semi-autonomous service still requires a great deal more testing, and further configuration will be required to fully specify how these processes should behave. Still, this takes the DCA, previously relying much more on direct manual management at every step by a system administrator, to a level of automation that, while still far from true autonomy, effectively reduces the time and effort required to run the service.

The work done for this project also creates a basis for continuing development of the system that will bring greater degrees of semi-autonomous behavior not only to the DCA service but to the system as a whole. The service daemon itself, designed to be generic and with reusable code, is ready to be deployed to all system services, along with the iDCA's monitoring capabilities, which will provide a system-wide view of available resources. Upon completion of this project, the system has also gained the benefit of having all its existing services installed and initialized on the basis of a single master configuration, though the completed configuration, beyond the initial testing phase, remains to be developed further.

In addition to this, while some of the features implemented for this project, such as request handling, remain at a very basic stage with much additional work to be done, and others, such as configuration checking, could be greatly extended and improved, the establishment of the basic mechanisms for these behaviors is an important first step for realizing their full potential.

At the conclusion of this project, another important result of this work is to highlight the importance of configurations to the desired semi-autonomous behaviors. While the

mechanisms now exist for the service daemon to monitor and respond to various events and messages, the key task remains to fully define all such expected events, so that they can be recognized effectively, along with defining the actions that they should trigger.

In the end, the benefit of these mechanisms for acting on monitored and received event data will be more fully realized through the ongoing and planned development of additional system services, including those that will handle alerting, more complex actions, and more complex processing of individual or combined event streams, in order to enable more sophisticated situational awareness and self-management in the system.

5 Discussion

The result of this project is a service with semi-autonomous behaviors, although it is less specifically focused than anticipated in the original project concept. The choice to orient the project less specifically toward the DCA and more toward application throughout the system means that some features from which a semi-autonomous DCA would benefit require significant further development work, whereas other features applicable throughout the system were developed further than originally anticipated.

During the course of the work for this project, a number of difficult decisions needed to be made that would impact the direction of the system as a whole. In each case, the different possible decisions posed different challenges that needed to be dealt with. In the end, some of these challenges remain as an indication of potential future development and improvement of the basis for semi-autonomous microservices developed during the course of this project.

5.1 Remaining Challenges and Potential for Improvement

One important area in which this project leaves room for potential improvements is in the monitoring of the DCA service. It remains at a basic level and without detailed insight into the internal processes of the service. While on one level this was an intentional decision, more insight could be gained by leveraging some of the new features developed for this project. For instance, the established inter-process interface could be used more effectively for exchanges between the DCA and the daemon to provide more specific service status information. This would require further changes to the DCA controller, which currently remains quite independent, shares its states indirectly, and does not make use of this direct line of communication.

While the structure of the service daemon and iDCA as service-neutral entities was an important step forward for the system, the need for additional customization in the form of specialized iDCA modules also remains as a key challenge. This was outside the scope of the present project, but it will be a necessary step going forward. Different types of specialized modules are needed for checking the status of different system services, as each functions quite differently, and the determination of their correct or abnormal functioning depends greatly on the different behavioral characteristics of each service.

For instance, the successful operation of the message bus depends upon its own proper configuration. If a port, for instance, has been misconfigured, or security settings are not aligned with those used by a consumer or producer of messages, the service could not be considered to be functional, even if the processes that constitute its core components are executing. For this functionality, it remains necessary to have a system administrator and programmer who knows the system and can create a custom solution. This also poses a challenge for expandability of the system, as any new services will require new status-checking modules.

Another place where this work is susceptible to continued improvement is in its capacity for checking new configurations during an update. This feature already utilizes the systemd facilities for monitoring a service to ensure that the service runs with its new configuration. However, this only ensures that the program itself can execute, which, as mentioned, does not guarantee that it is correctly configured to perform its assigned tasks.

A more sophisticated configuration verification mechanism should be able to recognize whether configurations are formatted correctly and whether anything is missing from them. It should then also be able to monitor the behavior of the service, after it begins running, to see not only that its processes are alive but that it is in fact functioning as intended. These changes can all be implemented in time through continuing development.

5.2 Future Expansion

The materials reviewed in the section on theoretical background above attest to the widespread focus on achieving degrees of autonomy in software, as well as specific efforts to incorporate self-monitoring and self-healing abilities into products. While this project moves the system forward significantly toward achieving such functionality, it also helps to bring into focus different possibilities for further enhancement and expansion in this direction.

First, in discussing the initial goals for this project, it was noted that a key aim of the project should be to create reusable solutions. The project’s success in doing this, through the creation of the common service daemon in particular, points to additional

reuse through further development of other system services as an important way of reaping additional benefits from this work.

As depicted in the system diagram (figure 5) included earlier, development on some of these relevant services was underway concurrently with this project, and other services are planned. In particular, the planned Stream Processor and Alerting/Action Handler services will contribute to the goals of increased autonomy in the system by contributing useful input that will enhance the daemon's self-monitoring and self-healing capabilities. The former will allow more complex stream analysis, including comparison of data across different event streams, while the latter will incorporate more advanced functionality for informing users about significant system events and executing complex actions that may span multiple services to address issues.

Another area in which future work on the system could better realize the goal of increased autonomy is by adding additional complexity to the new features of the semi-autonomous DCA. In particular, the ability of the DCA to alter its own configuration and that of its service daemon would be even more useful, and more beneficial to users, if it could be used automatically in a way that leverages intelligence gained from the DCA's own operation.

Capitalizing on the inherent benefits of its modular structure, adding the ability to reconfigure itself to the DCA will allow it to truly go beyond merely monitoring and maintaining its own behaviors to fundamentally altering those behaviors as well. Within the scope of the present project, configuration changes of that kind have been limited to altering behaviors or introducing new functionality through processes initiated and specified by human administrators. This limitation could be removed however through the use of machine learning, for example.

If the DCA could become able to intelligently adapt its own configuration, either independently or through communication with the larger system, to autonomously add and remove modules to suit the environments where it is deployed or the conditions its modules observe, the added power and removed need for user-customized configuration updates could be extremely powerful. This would build significantly on the concept, foundational to this project, of designing software once with the goal of deploying it in a range of differing contexts.

Yet another possibility for taking this project further through expansion based on existing features would be in the area of what Lemoine et al. refer to as “self-simulation” [16, p. 44], a property that was also deemed essential by Rajagopalan et al. [19]. This would essentially mean that a service would gain the ability to test other versions of itself while its current (known working) instantiation continues to execute, ensuring that there is as little downtime as possible for the service.

As it stands, this project is not capable of the kind of parallelism involved in true self-simulation, yet the MSA-based design of the system provides a path toward implementing this. It would be achieved much in the same way that the app-bisect project described by Rajagopalan et al. implemented a branching reverse-search for working configurations through parallel operation of microservices [19]. The capability of adding and removing copies of existing services with changed configurations, testing them without taking system components down or causing unwanted breaks in functionality, and preserving successful reconfigurations would enable a significantly more sophisticated level of autonomy for the system.

6 Conclusion

Achieving its original goal, this project documents the successful design and implementation of a semi-autonomous microservice. The definition of this goal at the outset of the project intentionally left room for a range of successful outcomes. The question of what “semi-autonomous” should mean in the context of this particular microservice was not entirely obvious at first, but it became clearer as the necessary decisions documented at each step of the project were made. In the end, this process contributed one important finding of the project, that automation is not a one-size-fits-all endeavor.

Automation and autonomous software come in many forms. They offer many different features and range in size and complexity from small-scale implementations, to reactive agents following complex rule sets, to advanced, proactive artificial intelligence. At the conclusion of this project, the semi-autonomous DCA implements a rule-following model of autonomous behavior. By adhering closely to its focus on maintaining a functional, yet reconfigurable system, it realizes its aims of adding value to the larger system and providing utility to users. In addition to this, it simplifies complicated managerial tasks and eliminates the need for constant vigilance on the part of system administrators.

The new DCA features developed over the course of this project do not aspire to realize full autonomy. Rather, they illustrate the powerful benefits that can be gained with even a modest degree of autonomous functionality. At the same time, they leave open the possibility for continuing improvements beyond the scope of the present project. In this, perhaps the project’s more important outcome is to demonstrate that automation can, and often should, be an ongoing process, with new opportunities to automate always emerging and new benefits accruing. In the end, these new DCA features, and their effects on the larger system, point to further possibilities for semi-autonomous control, some of which were not apparent when work on the project began.

In addition to providing these insights, this project also served as a valuable exercise in planning, particularly for additions to one part of a larger system. While more loosely coupled microservices can be developed more independently compared to layers of a monolithic application, it is nevertheless the case that a system-wide perspective must be maintained when considering how new features will be implemented. This project underscored the importance of considering how future possibilities for development will be constrained by every decision—even one as seemingly inconsequential as the format

used for a configuration file, which proved to have wide-ranging impacts across the system.

In the end, this project illustrates more broadly that decisions about the application of automation and the design of autonomous agents determine the nature of their impact on software systems, their users, and society as a whole. Autonomous software promises powerful advances in efficiency and productivity for users and administrators, but the degree to which this is achieved will depend on the thought and the breadth of perspective that go into their development and the care with which they are introduced. While a data collection agent is one small, less disruptive example of this emerging new paradigm, the same lesson applies even more to the major developments ahead. The challenges of autonomous software will undoubtedly grow as its capabilities increase, but there is reason to be optimistic about its potential to reshape human activities for the better.

References

1 Szymkowski S. An autonomous robot named CARL will charge your electric car. CNET [Internet]. 17 April 2020 [cited 27 August 2020]; Available from: https://www.cnet.com/roadshow/news/autonomous-robot-electric-car-charge-carl/

2 Barber G. This plane flies itself. We went for a ride. Wired [Internet]. 20 August 2020 [cited 27 August 2020]; Available from: https://www.wired.com/story/autonomous-plane-xwing/

3 Vincent J. Amazon expands its robot delivery trials to more states. The Verge [Internet]. 21 July 2020 [cited 27 August 2020]; Available from: https://www.theverge.com/2020/7/21/21332374/amazon-autonomous-robot-delivery-scout-expands-trials-atlanta-georgia-franklin-tennessee

4 Goodwin A. Survey: 3 out of 4 Americans say driverless cars 'not ready for primetime'. CNET [Internet]. 18 May 2020 [cited 27 August 2020]; Available from: https://www.cnet.com/roadshow/news/pave-autonomous-vehicle-survey/

5 Ahuja AS. The impact of artificial intelligence in medicine on the future role of the physician. PeerJ [Internet]. 4 October 2019 [cited 27 August 2020]; 7 e7702. Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6779111/pdf/peerj-07-7702.pdf DOI: 10.7717/peerj.7702

6 Clark K. Microservices vs SOA [web streaming video]. IBM Community; 1 April 2020 [cited 27 August 2020]. Available from: https://www.youtube.com/watch?v=KcR9mBLKFII

7 Hermann M, Pentek T, Otto B. Design principles for Industrie 4.0 scenarios: A literature review. Dortmund (DE): Technical University of Dortmund; 2015. 15 p. Working Paper No. 1 / 2015. Available from: https://pdfs.semanticscholar.org/069c/d102faebef48fbb7b531311e0127652d926e.pdf

8 Wolff E. Microservices: Flexible software architecture [Internet]. Addison-Wesley; 2016 [cited 27 August 2020]. Available from: https://learning.oreilly.com/library/view/Microservices:+Flexible+Software+Architecture/9780134650449/ch01.html#ch01lev1sec1

9 Mikkelsen A, Grønli T, Tamburri DA, Kazman R. Architectural principles for autonomous microservices. Proceedings of the 53rd Hawaii International Conference on System Sciences [Internet]. 2020 [cited 27 August 2020]. Available from: https://scholarspace.manoa.hawaii.edu/bitstream/10125/64546/0649.pdf

10 Newman S. Building Microservices. Sebastopol, CA: O'Reilly; 2015. 400 pg.

11 Baresi L, Ghezzi C, Guinea S. Towards self-healing composition of services. In: Krämer BJ, Halang WA, editors. Contributions to ubiquitous computing: Studies in computational intelligence, vol 42. Berlin: Springer; 2007. pp 27-46.

12 Dinh-Tuan H, Beierle F, Rodriguez Garzon SR. MAIA: A microservices-based architecture for industrial data analytics. IEEE [Internet]. 2019 [cited 27 August 2020]; Available from: https://arxiv.org/pdf/1905.06625.pdf

13 Hassan S, Bahsoon R. Microservices and their design trade-offs: A self-adaptive roadmap. IEEE [Internet]. 2016 [cited 27 August 2020]; Available from: https://ieeexplore-ieee-org.ezproxy.metropolia.fi/stamp/stamp.jsp?tp=&arnumber=7557535

14 Unland R. Software agent systems. In: Leitão P, Karnouskos S, editors. Industrial agents: Emerging applications of software agents in industry. Amsterdam: Elsevier; 2015. pp. 3-22.

15 King R. Digital workforce: Reduce costs and improve efficiency using robotic process automation. Self-published; 2018. 264 pg.

16 Lemoine F, Aubonnet T, Simoni N. Self-assemble-featured Internet of Things. Future Generation Computer Systems [Internet]. 13 May 2020 [cited 27 August 2020]; 112: 41-57. Available from: https://www-sciencedirect-com.ezproxy.metropolia.fi/science/article/pii/S0167739X20302843

17 Scully-Allison C, Le V, Fritzinger E, Strachan S, Harris FC, Dascalu SM. Near real-time autonomous quality control for streaming environmental sensor data. Procedia Computer Science [Internet]. 2018 [cited 27 August 2020]; 126: 1656-1665. Available from: https://www.sciencedirect.com/science/article/pii/S1877050918314170/pdf

18 Koch M, Pauls K. Engineering self-protection for autonomous systems. In: Baresi L, Heckel R, editors. Fundamental approaches to software engineering: Lecture notes in computer science [Internet]. Berlin: Springer; 2006 [cited 27 August 2020]. 3922: 33-47. Available from: https://link.springer.com/content/pdf/10.1007%2F11693017_5.pdf

19 Rajagopalan S, Jamjoon H. App-bisect: Autonomous healing for microservice-based apps [Internet]. New York: IBM T. J. Watson Research Center; 2019. Available from: https://www.usenix.org/system/files/conference/hotcloud15/hotcloud15-rajagopalan.pdf

20 Bradshaw JM. An introduction to software agents. In: Bradshaw JM, editor. Software agents. Menlo Park, CA: AAAI Press/MIT Press; 1997. pp. 3-46.

21 Ribeiro L. The design, deployment, and assessment of industrial agent systems. In: Leitão P, Karnouskos S, editors. Industrial agents: Emerging applications of software agents in industry. Amsterdam: Elsevier; 2015. pp. 45-63.

22 Shevat A. Designing bots: Creating conversational experiences. Sebastopol, CA: O’Reilly; 2017. 348 pg.

23 Foot P. The problem of abortion and the doctrine of the double effect. Oxford Review [Internet]. 1967 [cited 27 August 2020]; 5: 5-15. Available from: https://philarchive.org/archive/FOOTPO-2

24 Shams Z, De Vos M, Padget J, Vasconcelos WW. Practical reasoning with norms for autonomous software agents. Cornell University: arXiv.org [Internet]. 28 January 2017 [cited 27 August 2020]; Available from: https://arxiv.org/pdf/1701.08306

25 Ubuntu Manuals [Internet]. Canonical; 2019. Systemd; [updated 2019; cited 27 August 2020]; Available from: http://manpages.ubuntu.com/manpages/cosmic/man1/systemd.1.html