Service Colonies: A Novel Architectural Style for Developing Software Systems with Autonomous and Cooperative Services

Thakshila Dilrukshi
The University of Melbourne
Victoria, Australia
imiyamohotti@unimelb.edu.au

Artem Polyvyanyy
The University of Melbourne
Victoria, Australia
artem.polyvyanyy@unimelb.edu.au

Rajkumar Buyya
The University of Melbourne
Victoria, Australia
rbuyya@unimelb.edu.au

Colin Fidge
Queensland University of Technology
Queensland, Australia
c.fidge@qut.edu.au

Alistair Barros
Queensland University of Technology
Queensland, Australia
alistair.barros@qut.edu.au

Abstract

This paper presents the concept of a *service colony* and its characteristics. A *service colony* is a novel architectural style for developing a software system as a group of autonomous software services cooperating to fulfill the objectives of the system. Each inhabitant service in the colony implements a specific system functionality, collaborates with the other services, and makes proactive decisions that impact its performance and interaction patterns with other inhabitants. By increasing the level of self-awareness and autonomy available to individual system components, the resulting system is increasingly more decentralized, distributed, flexible, adaptable, distributed, modular, robust, and fault-tolerant.

1 Introduction

The architecture of a software system defines its structure and behavior in the context of the environment in which the system operates. Software system architectures evolve to be compatible with changing customer demands. For example, enterprise application architectures have evolved from legacy mainframe-based software systems and service-oriented architectures to microservices architectures [1, 23, 30]. Cloud-based technologies accelerate this transition by capitalizing on their scalability, flexibility, security, cost-effectiveness, and monitoring capabilities. The challenges of modern software systems, however, concern the increasing complexity of managing distributed architectures in highly dynamic environments. Despite changes occurring in its environment, a software system must continue to operate reliably and efficiently, delivering its functionalities to the users. Therefore, it is desirable to allow systems to adjust their structures and behaviors at runtime based on the environmental changes to enhance the transparency, elasticity, resilience, and deployment management [5].

Due to substantial investments already made in software systems, manually migrating them to new architectural styles is often infeasible. Automated enhancement of existing systems to meet current and future requirements is a more reasonable and cost-effective alternative. However, ensuring continuous operation under varying conditions often requires extensive human testing and supervision, leading to significant costs for configuring, troubleshooting, and maintaining applications [2]. As a result, there is an increasing demand for automated management and monitoring of distributed systems to reduce costs while ensuring their robustness and quality.

Self-adaptive software systems emerged to reduce uncertainty and complexity in dynamic operating environments [2]. They aim to improve the reliability, availability, flexibility, and performance of systems [28]. A self-adaptive autonomous software system monitors its internal and external states, identifying when and how to reconfigure to manage new conditions and dynamically adapt the software architecture at runtime [28]. Developing a self-adaptive software system is challenging due to conflicting system goals, the need to monitor different quality of service (QoS) properties, and handling complex mechanisms of adaptation and their effects [3]. Once developed, self-adaptive systems can perform self-configuring, self-healing, self-optimizing, and self-protecting functions [6, 18].

With the increased demand for microservices-based systems, self-adaptive microservices have gained attention. Even though decentralized decision-making and structural modifications can increase the system's adaptability, the often practiced strategy in this space is a top-down architecture with centralized monitoring and decision-making components [22]. This approach can support the development of agent-based software systems with improved autonomy in individual sub-systems while achieving the overall system's goals [15, 16]. Existing self-adaptive microservice systems focus on monitoring and managing the adaptation control loop and self-healing properties of the system. The prominent adaptation strategy is reactive adaptation, where the adaptation solution applies when the problem occurs [5]. Structural changes in the application layer via service aggregations and splittings, as well as the

introduction and removal of communication links between system services, would increase the level of flexibility of self-adaptive systems, providing scope for autonomous adaptations of the system's architecture. However, instead of the application layer, the focus in self-adaptive microservices is often on the management of the infrastructure layer, aiming to optimize system performance. Additionally, aspects relevant to the requirements of edge computing and executing systems under resource-constrained environments [33] are essential characteristics of future software systems to enhance the quality of the services provided to end users.

This paper introduces *service colonies*, a software architectural style for developing systems as groups of autonomous software services that cooperate to fulfill the global objectives of the overall system, increasing the level of autonomy and intelligence of the distributed (micro)services-based systems. Each inhabitant service of a colony fulfills a specific part of the functionality of the overall system the colony implements. By following prescribed or acquired through learning rules and by interacting with other inhabitants, the services demonstrate a swarm-like global intelligence in adapting the individual services, their deployment in the environment, and the configuration of communication links between the services and the environment that ensures continuous effective and efficient performance of the system. Though individual services can have different roles, no system components are envisioned to implement centralized control over the behavior and configuration of the overall system. An inhabitant service of a colony can interact with other services and the environment. Through interactions of individual services with the environment, the system interacts with its users to receive tasks and return results, as well as the hardware, software, and network infrastructure in which the system is deployed to optimize the utilization of the available resources. The interactions between the services ensure complex functions composed of functionalities of the interacting services can be realized.

By monitoring their behavior and environment, the inhabitant services of a colony identify their own and the overall system's performance bottlenecks and resource constraints and automatically adapt themselves and their communication links to overcome these issues. Therefore, service colonies follow a proactive adaptation strategy where the actions are taken before problems occur. Consequently, a service colony monitors, adapts, and optimizes its behavior through self-diagnosis and adaptations of individual components based on environmental conditions. Thus, it self-manages and self-architects in a dynamic environment. Moreover, it is an autonomous, goal-oriented, and goal-directed system that aims to enhance the quality of service provided to its users. In this way, service colonies extend self-adapting software systems, aiming to reduce the costly human interventions typically required for continuous monitoring, troubleshooting, and application maintenance.

The remainder of the paper proceeds as follows. Section 2 discusses existing service-based software architectures. Then, Section 3 presents service colonies. Section 4 discusses the envisaged benefits of this new architectural style, while Section 5 identifies challenges associated with the development of systems as service colonies. Finally, Section 6 concludes the paper.
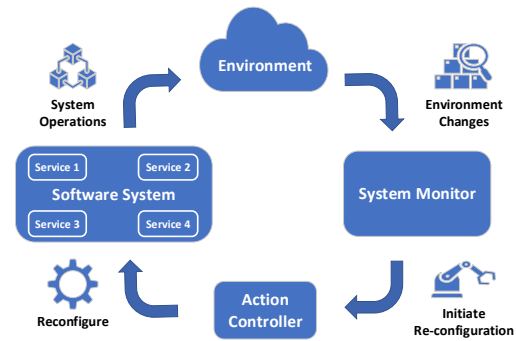

Figure 1: Self-adaptive system architecture

2 Related Work

In this section, we discuss three areas: (micro)service-oriented architectural styles [1, 23, 30], self-adaptive software systems [20, 21], and self-adaptive (micro)services-based systems [10, 12].

Service-oriented architectures [1] and microservices [23, 30] are software architectural styles that describe a system as a group of communicating services. Often, such a system identifies a primary service responsible for communications within the system. When a system receives a request, it delegates the necessary functions to its constituent services. While executing a specific functionality, a service can communicate with other services in the system. The response to the user is provided by aggregating the results of potentially multiple service calls.

A self-adaptive system architecture is illustrated schematically in Figure 1. A self-adaptive system can learn by observing the environment in which it is embedded. It has monitoring and decision-making components that are illustrated in Figure 1 as the system monitor component and the action controller component, respectively. The monitor component observes the environmental changes [28, 29, 35]. Once a significant change is identified, the monitor triggers the reconfiguration request. The action controller component identifies the required changes based on static rules or dynamically identified interventions. Subsequently, the action controller can initiate the reconfiguration of the system. Finally, the reconfigured system is deployed in the environment. Such self-adaptations are implemented as a repetitive, ongoing process.

The prominent technique for developing self-adaptive software systems is through control loop [3], which overviews the main activities of the system with a feedback cycle that starts with the relevant data collected from the system environment. The MAPE-K control loop [14, 18] monitors, analyzes, plans, and executes self-adaptive software systems [8, 26, 29]. Policy-based adaptations implement a static approach, where system experts analyze and design a policy repository for condition-triggered adaptations [7, 19]. These include pre-defined scenarios and corresponding actions to adapt the system. Rainbow [6], SASSY [27], MUSIC [13], and REACT [32] are example frameworks for software systems with self-adaptation control loops. However, these frameworks are not tailored towards decentralized systems of communicating services and require a profound knowledge of self-adaptive systems development [32].

Rainbow specifies an abstract architecture model to validate runtime properties of the system, evaluate them against the model of constraint violations, and perform global- and module-level adaptations of the system [6]. SASSY (Self-architecting Software Systems) is a model-driven framework that, based on pre-defined software adaptation patterns, aims to accommodate requirement changes in a dynamic environment. It generates candidate software architectures and selects the one that best serves stakeholder-defined, scenario-based quality-of-service (QoS) goals [27]. Finally, MUSIC provides a model-driven development approach combined with middleware that facilitates dynamic and automatic adaptation based on adaptation concerns [13]. SASSY, MUSIC, and REACT are all based on the MAPE-K control loop principles.

Self-adaptation in SOA has been discussed in the literature [9, 12]. Existing works propose middleware that supports cross-layer adaptation of SOA systems by considering the server-side perspectives [9]. This approach supports the adaptation of the service interface and application layer. It is based on a common metamodel of the two layers. Furthermore, dynamic self-adaptation in distributed service-oriented transactions [12] is an extension of the SASSY framework [27]. This extension proposes dynamic software adaptations in distributed transactions using a two-phase commit protocol. Specifically, it defines adaptation patterns and state machine models for transaction commits.

An automated approach for managing a collection of autonomic systems is based on the concept of a meta-manager that uses a parameterized adaptation policy [10, 11]. In this approach, a system is seen as composed of multiple constituent systems. Each constituent system contains a manager that keeps track of system metrics directly related to the overall system's quality of service objectives. Each manager selects the actions that will improve the system's performance. The approach implements a hierarchical control strategy that uses control theory to manage the behavior of the system while outsourcing decision-making responsibility to individual meta-manager units. However, this approach does not address the problem of structural changes in the application layer. Instead, it performs resource/capacity optimizations based on the runtime cost and response time analysis.

Based on the systematic mapping study [5], the state-of-the-art self-adaptive microservices-based systems focus on infrastructure layer adaptations [17, 25, 34] or multi-layer adaptations [9, 24, 37], while no studies address the problem of self-adaptation strategies in the application layer. Furthermore, there are works on reactive adaptation strategies, in which adaptations are applied after problems are identified [17, 25, 34]. Proactive adaptation strategies have been proposed for scientific workflows [31] and IoT architectures [4]. Centralized monitoring has been practiced as the prominent adaptation control mechanism [24, 34, 37].

The state-of-the-art studies on self-adaptive systems focus on the areas of cloud-based services, such as IoT and IaaS, rather than service-based software systems [36]. These approaches, which span across various domains (robotic, IoT, communication, automotive, e-commerce), utilize centralized monitoring, pre-defined models, and rule-based techniques [22, 36]. The introduction of self-managing and self-architecting principles in software systems could increase the sustainability of (micro)service-based architectures.

3 Service Colonies

A *service colony* is a system composed of software services, or *inhabitants*, that interact with each other to perform specific functions. The inhabitants are deployed in a distributed computing environment, such as a cloud, edge computing infrastructure, or IoT devices. Being intrinsically distributed, a service colony can exercise different levels of scatter in its control and decision-making capabilities across individual inhabitants, ranging from decentralized via divisional or hierarchical to centralized.

An *inhabitant* of a colony is responsible for performing some dedicated functionality, a *service*, within the system and exhibits a degree of autonomy. Hence, inhabitants within a colony may have diverse characteristics and capabilities, reflecting their different roles and responsibilities. Colony inhabitants have goals they aim to achieve and strive to optimize their performance and the performance of the overall system. Therefore, they are self- and situationally-aware entities. Inhabitants monitor their performance relative to their responsibilities, accepted service obligations, and the environment, and plan actions that ensure the desired quality of service. In doing so, they can proactively make decisions and take actions, such as replicating, migrating to another, more performant compute node, splitting into two entities, and seeking integration with another colony's inhabitants. An inhabitant may rely on learning mechanisms to improve their operations and decision rules based on prior experiences, allowing them to adapt and enhance their behavior.

An inhabitant can interact with other inhabitants and the environment. An inhabitant can request services from other inhabitants or provide service to them. In general, interaction patterns in a service colony can range from simple message exchanges between two inhabitants to complex collaboration protocols involving multiple inhabitants and the environment. Through these interactions, inhabitants coordinate activities, exchange information, and take inputs from and return outputs of the system to the users.

The environment of a service colony comprises its inhabitants, the computing infrastructure utilized by the colony, and the communication channels that the colony inhabitants can exploit. As special components in the environment, we identify the users of the overall colony system. The users seek to accomplish tasks and functions by interacting with interface entities of the colony and receiving results of such requests. The users seek to accomplish tasks and functions by interacting with interface entities of the colony and receiving the results of such requests. The environment of a service colony is, thus, inherently dynamic and stochastic.

As opposed to a self-adaptive system, a service colony has decentralized monitoring. In this proposed architecture, all the inhabitants act as autonomous agents. There is no special inhabitant responsible for overall communication, monitoring, or dynamic adaptations. Instead, each inhabitant can learn from its behavior and initiate actions to adjust the system automatically. Each individual inhabitant of a service colony can be (sub-)optimal, but collectively, they aim to optimize the overall system performance. Figure 2 sketches an example composition of service inhabitants in a service colony. Arcs between inhabitants indicate the links used to support communications between the individuals. Inhabitants can communicate with other inhabitants, for example, via messages
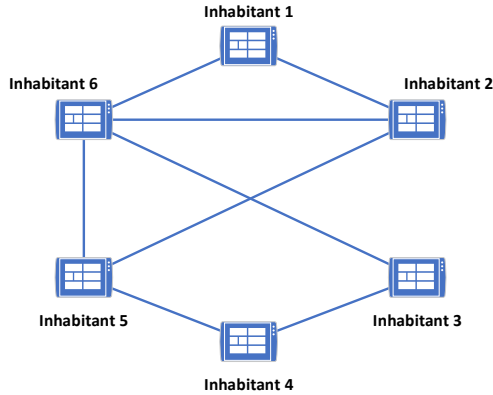
Thakshila Dilrukshi, Artem Polyvyanyy, Rajkumar Buyya, Colin Fidge, and Alistair Barros

Figure 2: An example service colony architecture

Figure 3: Inhabitant architecture

passing through communication links. Analytical messages and behavioral messages are the two types of special messages sent to the environment. Each inhabitant can initiate a change in the colony based on its analysis of itself and the environment using analytical messages. These can involve an inhabitant splitting, merging with another inhabitant, or adding or removing communication links between inhabitants. Furthermore, an inhabitant can send behavioral messages to the environment, indicating its capabilities or limitations. Examples of behavioral messages include the availability of more resources to be occupied by another inhabitant, delays in requests or responses, or the volume of data in requests or responses.

3.1 Inhabitant Characteristics

An inhabitant is the core building element of a service colony system. It encapsulates a delegated service, part of the functionality of the overall system, and monitors and proactively optimizes its performance. It is a self-contained entity. Each inhabitant has a boundary and a designated functionality that can be clearly distinguished from other inhabitants. Inhabitants are autonomous agents that can act and react independently. Inhabitants have a dynamic state that can change over time. Inhabitants have a state that can change over time. The future actions taken by an inhabitant depend on its current state. Inhabitants can share their state with other inhabitants in the colony. It is intended that an inhabitant frequently communicates and delivers services to other inhabitants. Thus, the current state of an inhabitant can be influenced by the state of another inhabitant in the colony and its behavior. Inhabitants are exploratory, and they can learn and adapt to the environment. They can learn from the environment and adapt their behavior based on their experience. Therefore, inhabitants can proactively adjust the system based on their behavior. Each inhabitant has goals, making it goal-directed and goal-oriented. An inhabitant attempts to accomplish these goals via its behavior and optimize the individual and overall system objective via adaptive learning.

3.2 Inhabitant Architecture

Figure 3 depicts the architecture of an inhabitant. Each inhabitant has a boundary and can send messages to and receive messages from the external environment. The sub-modules of an inhabitant include system functionality, communication, knowledge, and
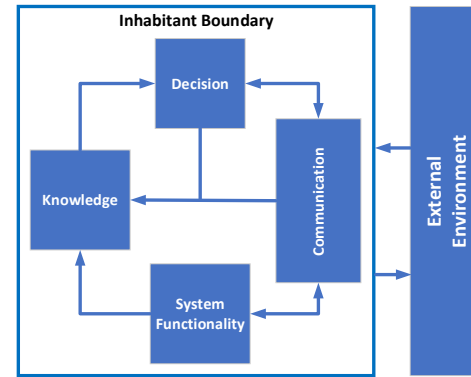
decision-making. These sub-modules communicate based on their responsibilities to provide the functionalities of the inhabitant.

The *communication* sub-module manages all interactions of the inhabitant and serves as its entry and exit point. Messages received from the environment, including those from other inhabitants, are captured by the communication module and forwarded to the relevant sub-modules. Messages initiated by the inhabitant and addressed to the environment are also routed through the communication sub-module for dissemination to their recipients. Upon receiving a message, the communication module validates its content to determine which sub-module is responsible for further processing. For example, functionality-related messages, such as requests to perform a function, are directed to the system functionality sub-module. Messages concerning environmental behavior are forwarded to the knowledge sub-module, while those pertaining to configurations and adaptations of inhabitants are redirected to the decision sub-module. Messages sent to the environment are routed to their intended recipients. Analytical messages can target individual inhabitants, subsets of inhabitants, or all inhabitants within the service colony. Behavioral messages are delivered to colony inhabitants based on the communication links within the topology of the colony.

The service colony has obligatory system functionalities that are distributed across its inhabitants. The *system functionality* sub-module is responsible for executing the inhabitant's delegated functionalities. Each inhabitant collaborates with other inhabitants in the service colony to fulfill these obligations. After completing a delegated function, an inhabitant responds to the inhabitants or users in the environment who have requested the result of the function via the communication module. Additionally, it sends log data to the knowledge sub-module to collect internal behavioral information.

Inhabitants monitor themselves and their environment. The *knowledge* sub-module builds the inhabitant's intelligence that aggregates the experiences of its monitoring processes. During initialization, an inhabitant collects data related to the service colony's structure and behavior. This information includes the total number of inhabitants, their configurations, and initial behavioral data of inhabitants. During execution, data and experiences stem from two sources. Functional data is collected from the system functionality sub-module. Additionally, an inhabitant gathers behavioral and analytical data of the service colony via the communication
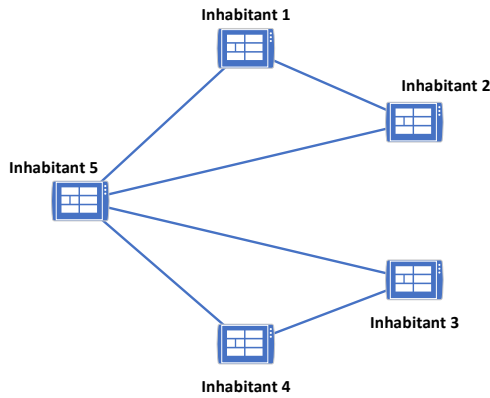
Figure 4: Service colony after merging

Figure 5: Service colony after splitting

sub-module. The inhabitant does not accept all the data received from the environment. It filters data based on quality and relevance. Furthermore, it tracks the origin of the data for the decision-making process executed by the decision sub-module.

Inhabitants can react to changes in their environment. They learn and take action based on their behavior and the evolution of the environment. The *decision* sub-module handles these learning and decision-making functions. This sub-module depends on data provided by the knowledge sub-module. Learning is driven by data available in the knowledge sub-module and previous decisions made by the decision sub-module. An inhabitant can use adaptive learning to dynamically adjust to the environment based on the current and past behavior of the system. Data collected after implementing a decision is used as feedback for learning. The decision sub-module handles two functions. First, if data from the inhabitant indicates a relevant behavior, it disseminates that message to the environment via the communication sub-module. These are behavioral messages. Second, it analyzes inhabitant and environment data to identify system operations, such as operations relevant to rearchitecting the system, and actions for improving performance. Such information is shared with other inhabitants in the colony via analytical messages. Each inhabitant has objectives captured as collections of rules within the decision sub-module. These rules can be updated dynamically via the interface provided to the decision sub-module. Therefore, during execution, new rules can be added, while existing rules can be modified or deleted.

3.3 Operations

An inhabitant can replicate itself. If an inhabitant experiences a high volume of requests and no further optimizations of the inhabitant are possible, it can decide to replicate itself to accommodate the high volume of service requests.

A service colony can change its structure to adapt to the environment and optimize its communication patterns. Four basic operations are proposed to support such adaptations: joining two or more inhabitants, splitting an inhabitant into two or more inhabitants, and adding and removing communication links between inhabitants. Two or more inhabitants can *join* to form a single inhabitant, for example, to simplify the communication structure between themselves and the rest of the colony during a low system load. Alternatively, such joining can be triggered to maximize the
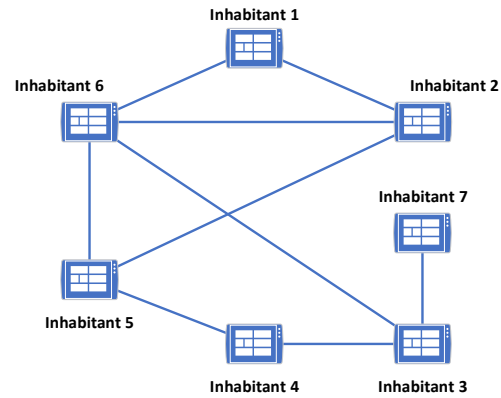
resource utilization in the system, hence reducing the operation cost. A complex join of multiple inhabitants can be implemented as a sequence of atomic joins between pairs of inhabitants. That is, two inhabitants merge in the first step. Then, another inhabitant joins with the previously merged inhabitant. An inhabitant can *split* into multiple inhabitants via a sequence of atomic splits of one inhabitant into two inhabitants to distribute its functionality and responsibilities among multiple system elements. Such splittings can be initiated based on the identified performance bottlenecks and resource-intensive functionalities that degrade the quality of the service provided. Hence, it can split its functionalities to maximize performance, for instance, by replicating those decomposed services that receive high request loads. Finally, an inhabitant can establish direct communication links with other inhabitants, for example, to reduce current communication latency or to identify peer inhabitants from which to request required services. Moreover, merging and splitting operations in the system could lead to the addition and removal of communication links in the colony.

Figure 4 depicts the re-architectured system from Figure 2 after merging inhabitant 6 with inhabitant 5. This operation leads to redirecting the communication links of inhabitant 6 to and from the resulting inhabitant 5. An example splitting of inhabitant 3 is depicted in Figure 5. In this case, inhabitant 7 is split out of the original inhabitant 3 in the system. In this example, a communication link is established between the resulting inhabitants. The inhabitants can use this link to request services from each other.

One can come up with different strategies for deciding which adaptations of the system structure to perform. Such strategies can result in the authoritative execution of the intended adaptation or an adaptation after confirmation from peers, confirmation from a group of inhabitants, or confirmation from the entire service colony. Inhabitants execute authoritative actions without requesting confirmation from other members of the colony. Alternatively, an inhabitant can negotiate the intended splittings and joins with other inhabitants to ensure mutual agreement and maximal benefit for all the negotiators.

Inhabitants can request confirmation from their peers in case the change affects them. Similarly, if the change affects a group of inhabitants of the colony, permissions can be requested from the impacted inhabitants. For instance, if inhabitant 6 from Figure 5 decides to split into two services, it may require obtaining permission
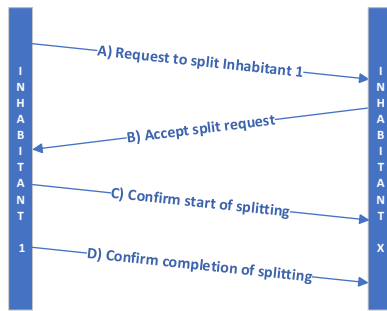
Figure 6: Splitting request communication sequence

from inhabitants 1, 2, 3, and 5 since they have direct communication links with inhabitant 6. Execution of an adaptation decision is initiated by sending a message to the relevant inhabitants to request permission. Once these requests arrive at the recipients, they validate the requested change. The validation is based on the knowledge stored within the decision sub-modules. Once validated, the inhabitants reply with approvals or rejections of the change. If the change is approved, the requesting inhabitant executes the change process. At the beginning of this process, the inhabitant confirms the start of the change to the environment. After the inhabitant receives a confirmation to start the process, it should not accept further adaptation requests until the current execution process is completed and confirmed. Then, the inhabitant executes the change. After completing the change, it informs the environment by sending the execution completion message. Then, other inhabitants update their configurations based on the executed change. These updates can involve adding, removing, or updating new or existing links between inhabitants.

Figure 6 illustrates an example communication sequence for splitting an inhabitant in a service colony. An inhabitant requests the environment to split. After other inhabitants accept the request, the splitting task is confirmed, and the splitting process is executed. After successful splitting, the confirmation regarding the successful splitting is sent to the environment. Then, the relevant inhabitants of the colony update their knowledge and links. If the inhabitant's request to split is rejected, no further actions are taken, and the communication ends. Figure 7 illustrates the scenario of merging two inhabitants. In this case, inhabitant 2 identifies the need to merge and acts as the leading inhabitant of the change process. First, it seeks to get confirmation from inhabitant 1, the inhabitant it intends to merge with. Once inhabitant 1 accepts the merging request, the next request from inhabitant 2 to the environment is sent to get approval to merge. If the environment responds positively, the merging process commences. After the merging process is completed, the execution confirmation message is sent to the environment to update relevant configurations.

4 Benefits

Both service-oriented architecture and microservices architecture identify loosely coupled services that can be deployed separately from the main application. These individual services can be independently designed, developed, and tested. Furthermore, using
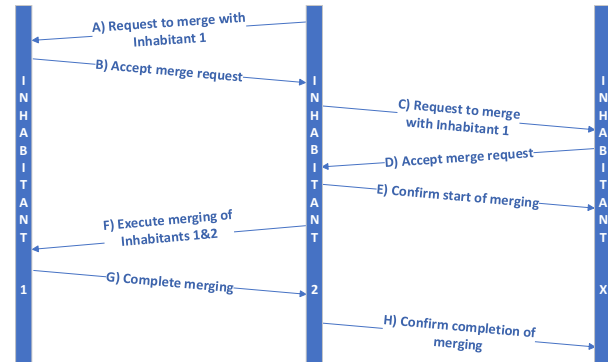

Figure 7: Merging request communication sequence

containerization techniques, services can auto-scale in a cloud-based environment based on their performance. In general, a self-adapting system handles specific domain requirements and resource optimizations and adaptations in the infrastructure layer using reactive error-handling functionality executed in pre-modeled situations [22, 36]. Service colonies, however, are not restricted by pre-identified or pre-modeled scenarios. A colony can proactively re-architect itself using a bottom-up approach and decentralized analysis of the system.

Consider an online shopping system developed as a service-based application. Assume that user registration and shopping cart actions are developed as two separate services within the system. In December, due to the festive season, the system receives a 100% surge in the volume of requests. During this high workload period, shopping cart services can experience performance bottlenecks, for example, due to latency in the payment handling process. A standard way to address this scenario is to increase resources for the entire shopping cart actions service. If this system is implemented as a service colony, it can identify that the bottleneck is caused by the payment transactions and split out this functionality into a new service. Subsequently, the colony can aim to utilize available resources to scale the payment transactions, for instance, by replicating the new payment service while keeping other shopping cart actions within the original service. These nuanced adaptations can drastically reduce the operating costs of the system. Moreover, when the system experiences a low volume of requests, the shopping cart actions and payment handling services can merge back to achieve a smaller system footprint. Note that within a service colony, such splitting and merging happen automatically, reducing the costs of maintaining the system.

The benefits of a service colony are, thus, at least the following:

- **Extension of Microservices.** Service colonies build upon the strengths of microservice-based architectures, inheriting their flexibility, resilience, modularity, optimized resource usage, and reduced operational overhead. Since the industry is moving toward broader adoption of microservice-based systems, service colonies contribute to this trend, providing more options for engineering future software systems.

- **Enhanced Flexibility.** By fostering interactions among individual inhabitants, a service colony ensures that desired functionalities are delivered flexibly. For example, multiple inhabitants can

deliver the same functionality based on different service agreement levels or with improved performance by proactively scaling or migrating the functionality to more productive compute nodes. In addition, each inhabitant can be developed, deployed, tested, and updated independently.

- **Proactive Fault Tolerance.** Each inhabitant in the service colony generates its own analytics and shares them throughout the colony. The individual and collaborative analysis of this data supports effective predictions of the system's behavior, including potential future faults. Since the analysis is based on the runtime information of individual services, it can be used to effectively monitor, predict, and proactively respond to environmental changes and system faults.

- **Continuous Optimization.** Through continuous learning from its operations, each inhabitant, and consequently the entire service colony, evolves into a self-optimizing entity, constantly refining itself for enhanced performance.

- **Goal Orientation.** Both the service colony as a whole and its individual inhabitants have objectives and strive to achieve them. The collaborative nature of the system fosters iterative improvements toward shared goals.

- **Dynamic Service Introduction and Composition.** The dynamic introduction and integration of services within a service colony optimize system performance and resource utilization, seamlessly adapting to evolving demands and environmental changes.

- **Heterogeneity.** A service colony hosts diverse inhabitants, each potentially utilizing different technologies and implementations. However, adherence to standard communication protocols ensures effective data dissemination throughout the ecosystem.

- **Scalability.** By facilitating individual scalability based on performance metrics, a service colony can reduce maintenance costs through proactive decision-making and predictive scaling initiated by its inhabitants.

5 Challenges

The design, implementation, and maintenance of a service colony come with challenges that are yet to be understood and studied. We initiate this endeavor by discussing several such challenges below:

- **Complex System.** A service colony is a dynamic, complex system composed of interacting components. Without hierarchical control or global coordination, in general, slight modifications of components and interaction patterns can have a substantial impact on the overall high-level behavior of the system. Without hierarchical control or global coordination, even slight modifications to components and interaction patterns can significantly impact the system's overall behavior. This intricate relationship between low-level component behaviors and high-level system behavior complicates error identification and troubleshooting. Additionally, if the splitting of inhabitants is not controlled, the colony can proliferate excessively, increasing system latency. Although the system aims to optimize for latency, unnecessary adjustments can result in the opposite effect.

- **Verification and Validation.** To verify the correctness of a system, it is essential to test it under different conditions, for example, by simulating environmental changes. The distributed nature of a service colony complicates this process due to the vast number of possible scenarios the system can execute. New tools are required to simulate different workloads and environmental changes to properly test service colonies. Additionally, validating the correctness of system decisions before rearchitecting is challenging due to the system's dynamic nature, which makes it difficult to predict outcomes and ensure stability in advance.

- **Dynamic Updates.** A service colony is a dynamic system. Adjustments within such a system can lead to temporary unavailability of certain functionalities. Therefore, sophisticated mechanisms are needed to manage these dynamic adjustments effectively, ensuring smooth operations during system rearchitecting.

- **Heterogeneity.** A service colony can host diverse inhabitants, each implemented using different technologies. To support this heterogeneity, standard interfaces and communication protocols must be established. Moreover, the integration of components implemented using different technologies complicates system development, testing, and monitoring. Furthermore, such a heterogeneous system is more vulnerable to security threats.

- **Persistence.** A service-based system relies on databases and data caching layers for information storage. These components require non-trivial adaptations during the reengineering of the system. Dynamic repartitioning and redesigning of databases can lead to data replication, which may result in data inconsistencies.

- **System Updates and Change Requests.** System updates are mandatory to comply with industry standards, and customer change requests are inevitable. Implementing these changes in a distributed system is a complex task. A robust system state management process must be in place to handle updates effectively. Automatically persisting the system state before and after adjustments is essential for maintaining system operations. Additionally, a service colony must be equipped with comprehensive mechanisms for the automatic deployment of services resulting from the splitting of colony inhabitants and for managing continuous delivery and integration pipelines.

- **Reengineering.** Existing systems that wish to benefit from the advantages of service colony architecture need to be reengineered accordingly. A systematic process for reengineering software systems into service colonies must be defined to facilitate the migration of legacy systems to this new architectural style.

6 Conclusions

This paper introduces the concept of a *service colony*, a software architectural style for developing systems as groups of autonomous, interacting software services. Each inhabitant service in a colony is driven by its aim to deliver services to its users, either external users of the system or other inhabitants of the colony. Based on their past performance, inhabitants can proactively decide to self-replicate, split into multiple services, or join with other inhabitants to ensure high-quality service delivery. In this way, the overall service colony system can adapt to changing workloads by either shrinking its footprint during periods of low workload or scaling specific high-demand functionality during workload bursts. By performing such adaptations, the system aims to minimize resource utilization while

maximizing the quality of the delivered services over time. A service colony is a bottom-up complex system characterized by numerous interacting components that result in emergent system-level behavior. Consequently, service colonies aim to inherit the benefits of complex system architectures, including resilience, robustness, adaptability, scalability, and distributed control. Future work on service colonies will focus on designing and evaluating different colony inhabitant architectures and principles of their interactions to study their effects on the global behavior of the overall system.

References

[1] Charles Abrams and Roy W. Schulte. 2008. Service-oriented Architecture Overview and Guide to SOA Research. *Gartner Research* (2008).

[2] Ahmad Banijamali, Pasi Kuvaja, Markku Oivo, and Pooyan Jamshidi. 2020. Kuksa*: Self-adaptive Microservices in Automotive Systems. In *PROFES (LNCS, Vol. 12562)*. Springer, 367–384.

[3] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. *Software Engineering for Self-Adaptive Systems: A Research Roadmap.* LNCS, Vol. 5525. Springer, 1–26.

[4] Martina De Sanctis, Henry Muccini, and Karthik Vaidhyanathan. 2020. Data-driven Adaptation in Microservice-based IoT Architectures. In *ICSA Companion.* IEEE, 59–62.

[5] Messias Filho, Eliaquim Pimentel, Wellington Pereira, Paulo Henrique M. Maia, and Mariela I. Cortés. 2021. Self-Adaptive Microservice-based Systems: Landscape and Research Opportunities. (2021), 167–178.

[6] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer* 37, 10 (2004), 46–54.

[7] David Garlan, Bradley R. Schmerl, and Shang-Wen Cheng. 2009. Software Architecture-Based Self-Adaptation. In *Autonomic Computing and Networking.* Springer, 31–55.

[8] Omid Gheibi, Danny Weyns, and Federico Quin. 2020. Applying Machine Learning in Self-adaptive Systems: A Systematic Literature Review. *ACM Trans. Auton. Adapt. Syst.* 15, 3 (2020), 9:1–9:37.

[9] Eli Gjørven, Romain Rouvoy, and Frank Eliassen. 2008. Cross-layer Self-adaptation of Service-oriented Architectures. In *MW4SOC.* ACM, 37–42.

[10] Thomas J. Glazier and David Garlan. 2019. An Automated Approach to Management of a Collection of Autonomic Systems. In *FAS*W@SASO/ICAC.* IEEE, 110–115.

[11] Thomas J. Glazier, David Garlan, and Bradley R. Schmerl. 2020. Automated Management of Collections of Autonomic Systems. In *ACSOS.* IEEE, 82–91.

[12] Hassan Gomaa and Koji Hashimoto. 2012. Dynamic Self-adaptation for Distributed Service-oriented Transactions. In *SEAMS.* IEEE Computer Society, 11–20.

[13] Svein O. Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, and George Angelos Papadopoulos. 2012. A Development Framework and Methodology for Self-adapting Applications in Ubiquitous Computing Environments. *J. Syst. Softw.* 85, 12 (2012), 2840–2859.

[14] IBM. 2006. An Architectural Blueprint for Autonomic Computing. *IBM White Paper* (2006).

[15] Nicholas R. Jennings. 2000. On Agent-based Software Engineering. *Artif. Intell.* 117, 2 (2000), 277–296.

[16] Nicholas R. Jennings. 2001. An Agent-based Approach for Building Complex Software Systems. *Commun. ACM* 44, 4 (2001), 35–41.

[17] Stefan Kehrer and Wolfgang Blochinger. 2018. Model-Based Generation of Self-adaptive Cloud Services. In *CLOSER (Selected Papers) (Communications in Computer and Information Science, Vol. 1073).* Springer, 40–63.

[18] Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50.

[19] Narges Khakpour, Ramtin Khosravi, Marjan Sirjani, and Saeed Jalili. 2010. Formal Analysis of Policy-based Self-adaptive Systems. In *SAC.* ACM, 2536–2543.

[20] Jeff Kramer and Jeff Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Software Eng.* 16, 11 (1990), 1293–1306.

[21] Jeff Kramer and Jeff Magee. 2007. Self-Managed Systems: An Architectural Challenge. In *FOSE.* IEEE Computer Society, 259–268.

[22] Christian Krupitzer, Felix Maximilian Roth, Martin Pfannemüller, and Christian Becker. 2016. Comparison of Approaches for Self-Improvement in Self-Adaptive Systems. In *ICAC.* IEEE Computer Society, 308–314.

[23] James Lewis and Martin Fowler. 2014. Microservices: A Definition of This New Architectural Term. https://martinfowler.com/articles/microservices.html. Accessed on 7 July 2024.

[24] Peini Liu, Xinjun Mao, Shuai Zhang, and Fu Hou. 2018. Towards Reference Architecture for a Multi-layer Controlled Self-adaptive Microservice System. In *SEKE.* KSI Research Inc. and Knowledge Systems Institute Graduate School, 236–235.

[25] Basel Magableh and Muder Almiani. 2019. A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster. In *AINA (Advances in Intelligent Systems and Computing, Vol. 926).* Springer, 846–858.

[26] Lukas Malburg, Maximilian Hoffmann, and Ralph Bergmann. 2023. Applying MAPE-K Control Loops for Adaptive Workflow Management in Smart Factories. *J. Intell. Inf. Syst.* 61, 1 (2023), 83–111.

[27] Daniel A. Menascé, Hassan Gomaa, Sam Malek, and João Pedro Sousa. 2011. SASSY: A Framework for Self-Architecting Service-Oriented Systems. *IEEE Softw.* 28, 6 (2011), 78–85.

[28] Nabor Chagas Mendonça, Pooyan Jamshidi, David Garlan, and Claus Pahl. 2021. Developing Self-Adaptive Microservice Systems: Challenges and Directions. *IEEE Softw.* 38, 2 (2021), 70–79.

[29] Andreas Metzger, Clément Quinton, Zoltán Ádám Mann, Luciano Baresi, and Klaus Pohl. 2024. Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration. *Computing* 106, 4 (2024), 1251–1272.

[30] Sam Newman. 2015. *Building Microservices: Designing Fine-grained Systems.* O'Reilly.

[31] Phuong Nguyen and Klara Nahrstedt. 2017. MONAD: Self-Adaptive Micro-Service Infrastructure for Heterogeneous Scientific Workflows. In *ICAC.* IEEE Computer Society, 187–196.

[32] Martin Pfannemüller, Martin Breitbach, Christian Krupitzer, Markus Weckesser, Christian Becker, Bradley R. Schmerl, and Andy Schürr. 2020. REACT: A Model-Based Runtime Environment for Adapting Communication Systems. (2020), 65–74.

[33] Ali Rahmanian, Ahmed Ali-Eldin, Björn Skubic, and Erik Elmroth. 2022. MicroSplit: Efficient Splitting of Microservices on Edge Clouds. In *SEC.* IEEE, 252–264.

[34] Adalberto R. Sampaio, Julia Rubin, Ivan Beschastnikh, and Nelson S. Rosa. 2019. Improving Microservice-based Applications with Runtime Placement Adaptation. *J. Internet Serv. Appl.* 10, 1 (2019), 4:1–4:30.

[35] Danny Weyns, M. Usman Iftikhar, Danny Hughes, and Nelson Matthys. 2018. Applying Architecture-Based Adaptation to Automate the Management of Internet-of-Things. In *ECSA (LNCS, Vol. 11048).* Springer, 49–67.

[36] Terence Wong, Markus Wagner, and Christoph Treude. 2022. Self-adaptive Systems: A Systematic Literature Review Across Categories and Domains. *Inf. Softw. Technol.* 148 (2022), 106934.

[37] Shuai Zhang, Mingjiang Zhang, Lin Ni, and Peini Liu. 2019. A Multi-level Self-adaptation Approach for Microservice Systems. In *ICCCBDA.* IEEE, 498–502.