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

THAKSHILA IMIYA MOHOTTIGE, The University of Melbourne, Australia
ARTEM POLYVYANYY, The University of Melbourne, Australia
COLIN FIDGE, Queensland University of Technology, Australia
RAJKUMAR BUYYA, The University of Melbourne, Australia
ALISTAIR BARROS, Queensland University of Technology, Australia

This paper introduces the concept of a *service colony*, a novel architectural style for developing software systems. A *service colony* organizes a system as a group of autonomous software services that cooperate to achieve its objectives. Each service in the colony is responsible for implementing a specific functionality, interacting with other services and system users, and making proactive decisions that influence its and the system's performance, as well as the system's topology and interaction patterns. By enhancing the self-awareness and autonomy of individual components, this architecture fosters greater decentralization, flexibility, modularity, and robustness. Experiments involving the reengineering of an open-source system into a service colony demonstrate its ability to dynamically rearchitect and adjust performance in response to changing user demands and usage intensity.

CCS Concepts: • **Software and its engineering → Distributed systems organizing principles**.

Additional Key Words and Phrases: Software architectures; Component-based systems; Distributed systems; Self-modifying systems

1 Introduction

The architecture of a software system defines its structure and behavior within the environment in which it operates. Software system architectures evolve to meet changing customer demands. For example, enterprise application architectures have evolved from legacy mainframe-based software systems and service-oriented architectures (SOA) to microservices architectures (MSA) [1, 36, 47]. 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 its users. Therefore, it is desirable to allow systems to adjust their structures and behaviors at runtime based on environmental changes to enhance transparency, elasticity, resilience, and deployment management [17].

Authors' Contact Information: Thakshila Imiya Mohottige, The University of Melbourne, Melbourne, Victoria, Australia, imiyamohotti@unimelb.edu.au; Artem Polyvyanyy, The University of Melbourne, Melbourne, Victoria, Australia, artem.polyvyanyy@unimelb.edu.au; Colin Fidge, Queensland University of Technology, Brisbane, Queensland, Australia, c.fidge@qut.edu.au; Rajkumar Buyya, The University of Melbourne, Melbourne, Victoria, Australia, rbuyya@unimelb.edu.au; Alistair Barros, Queensland University of Technology, Brisbane, Queensland, Australia, alistair.barros@qut.edu.au.

ACM 1557-7392/2026/6-ART
https://doi.org/10.1145/3821209

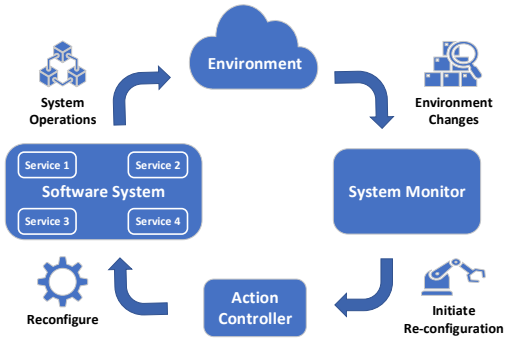


Fig. 1. A typical self-adaptive system architecture [19].

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 high costs for configuration, troubleshooting, and maintenance of applications [8]. 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 [7].

Self-adaptive software systems emerged to reduce uncertainty and complexity in dynamic operating environments [8]. They aim to improve the system's reliability, availability, flexibility, and performance [42]. A self-adaptive system architecture is illustrated schematically in Figure 1. 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 [42, 65]. It has monitoring and decision-making components, as illustrated in Figure 1, namely the system monitor and the action controller. The monitor component observes the environmental changes [42, 44, 64]. 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. Developing a self-adaptive software system is challenging due to conflicting system goals, the need to monitor different quality of service (QoS) properties, and handle complex mechanisms of adaptation and their effects [15]. Once developed, self-adaptive systems can perform self-configuring, self-healing, self-optimizing, and self-protecting functions [19, 33].

With the increasing demand for microservice-based systems, introducing self-adaptive and autonomous properties to microservices has attracted attention. The majority of studies focus on optimized auto-scaling in cloud environments, particularly CPU and memory scaling [28, 62]. In contrast, certain studies on self-adaptive microservice systems emphasize the monitoring and management of the adaptive control loop, as well as the system's self-healing properties. The predominant adaptation strategy is reactive adaptation, in which the adaptation solution is applied when the problem occurs [17]. Although 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 [35]. 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 [30, 31]. Furthermore, aspects relevant to edge computing and system

execution in resource-constrained environments are essential characteristics of future software systems to enhance the service qualities provided to end users [55].

However, these approaches do not address issues of architectural flexibility. Performance degradation may stem from overloaded responsibilities or inappropriate service decomposition rather than from a lack of computational resources [63, 68]. Adapting the architectural structure is necessary when services exhibit heterogeneous scaling behavior. Components within a system may exhibit different demand growth rates, making uniform scaling inefficient. For instance, a validation service may encompass both functional and security validation. Security validation is often more computationally expensive and subject to higher variability. Decoupling such concerns and optimizing them independently reduces resource waste and isolates performance variability. Moreover, system workloads evolve dynamically due to changes in user behavior, feature additions, and seasonal demand patterns. Structural adaptation enables systems to split services when increased concurrency or specialization is required, and to merge services when traffic decreases, and inter-service communication overhead dominates. Although microservice architecture improves modularity and scalability, they also introduce additional latency, coordination complexity, and operational costs. Under low-load conditions, service consolidation can reduce network overhead, deployment complexity, and monitoring costs. By enabling runtime adaptation of the architectural structure, systems achieve elasticity at the architectural level, rather than solely at the resource level. This approach improves fault isolation, mitigates cascading failures, supports continuous evolution, and improves overall cost efficiency [63, 68].

Accordingly, this paper introduces the notion of a *service colony*, 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 distributed (micro)services-based systems. Each inhabitant service of a colony fulfills a specific part of the functionality of the overall system that the colony implements. By following rules, either prescribed or acquired through learning, 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 may have different roles, no system components are envisioned to implement centralized control over the overall system's behavior and configuration. An inhabitant service of a colony can interact with other services and the environment. Through interactions with individual services, the system interacts with its users to receive tasks and return results, as well as with the hardware, software, and network infrastructure in which it is deployed, to optimize the utilization of available resources. The interactions between the services ensure that complex functions composed of the constituent services' functionalities can be realized.

By monitoring both their behavior and environment, the inhabitants 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, taking corrective actions 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, reducing the need for the costly human interventions typically required for continuous monitoring, troubleshooting, and application maintenance.

In this paper, we present a proof-of-concept of the effectiveness of service colonies. The evaluation focuses on split, merge, and multiple split/merge scenarios, as well as the behavior of service colonies in resource-constrained environments. Our experimental results indicate that service colonies improve response time and memory utilization of the system. The benefits are particularly pronounced in resource-constrained environments,

manifesting as improvements in throughput, average response time, adaptation efficiency, and memory consumption. These results suggest that service colonies are well-suited for IoT and edge computing environments, where computational and memory resources are limited.

The rest of the paper is organised as follows. Section 2 reviews existing service-based software architectures. Section 3 presents the concept of a service colony. Section 4 discusses our proof-of-concept implementation, self-adaptive description language of a service colony system, and reports experimental results. Section 5 explores the potential benefits of service colonies and examines the challenges in developing software systems as service colonies. Finally, Section 6 summarizes the contributions of the paper along with directions for future work.

2 Related Work

In this section, we review (micro)service-oriented architectural styles, self-adaptive (micro) services-based systems, and self-adaptive approaches practiced in the domain.

Service-oriented architectures [1] and microservices [36, 47] 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.

Service colonies relate to self-adaptive systems. Hence, a systematic review has been conducted to compare the service colonies with existing distributed component/(micro)service-based systems. Relevant studies were retrieved from the Web of Science[1] and Scopus[2] databases using the keywords "self-adapt*", "software*", and "architecture". The initial search yielded 1,496 publications. After successive screening and filtering steps, 21 relevant studies were identified.

A comparative analysis of reviewed studies is presented in Table 1. The characteristics presented in the table describe the key architectural and functional properties of the proposed system. Next, we explain these characteristics. A *self-adaptive system* is one that autonomously adjusts its behavior in response to environmental changes, workload variations, or internal performance metrics without requiring external intervention. *Decentralized decisions* imply that the control and decision-making are distributed across multiple components rather than relying on a single centralized controller. The *agent-based* property refers to the availability of autonomous agents that operate with local intelligence and collaborate to achieve global objectives. Operating at the *application layer* indicates that adaptation and coordination mechanisms are implemented above the network layer. A *learning-based* approach incorporates machine learning or data-driven techniques to improve the quality of decisions over time continuously. Being *proactive* means that the system anticipates future conditions and takes preventive actions instead of reacting to events. *Split/merge actions* refer to the ability of the system to dynamically divide or consolidate services to optimize performance and resource utilization. Finally, *generalizable* denotes that the proposed approach is not limited to a single use case but can be applied across different domains or scenarios.

The eight properties selected for comparison were derived from the reviewed studies. Each property represents a functional or architectural criterion considered in multiple reviewed studies, ensuring that the comparison reflects aspects that the research community has identified as relevant to self-adaptive service-based systems. Properties present in all reviewed studies, such as the feedback loop for adaptation, and highly system-specific properties, such as specific communication protocols and deployment infrastructure requirements, were excluded from the comparison because they are not generalizable across the heterogeneous set of reviewed studies. Properties that the service colony does not address, such as formal verification of adaptation correctness and efficiency, were

[1]https://clarivate.com/webofsciencegroup/solutions/web-of-science
[2]https://www.scopus.com/

Table 1. A comparison of existing self-adaptive software systems.

Study Ref	Self-adaptive system	Decentralized decisions	Agent-based	Application layer	Learning -based	Proactive	Split/ merge actions	Generalizable
[4]	●	●	○	○	○	●	○	●
[11]	●	○	○	●	○	○	○	○
MOSES [14]	●	○	○	○	○	○	○	●
SAFDIS [20]	●	●	◐	●	◐	●	○	○
S/T/A [27]	●	○	○	●	○	N/A	○	●
SAMSP [37]	●	○	○	●	○	○	○	●
SASSY [41]	●	○	○	●	○	○	○	●
[46]	●	●	●	●	●	●	○	○
[51]	●	○	○	○	○	○	○	○
[52]	●	○	○	●	○	○	○	●
[54]	●	○	○	●	○	○	○	○
[59]	●	◐	◐	●	●	○	○	○
DySOA [60]	●	○	○	●	○	○	○	○
[61]	●	●	●	N/A	●	○	○	●
EPF4M [63]	◐	○	○	●	○	○	●	●
[68]	●	○	○	●	○	○	◐	○
[2]	●	●	○	○	○	○	○	◐
DARE [3]	●	●	○	●	○	○	○	◐
[9]	●	●	◐	●	○	○	○	●
[25]	●	○	○	●	○	○	○	○
[45]	●	◐	○	●	○	○	○	●
Service Colonies	●	●	●	●	●	●	●	●

● Addressed ◐ Partially addressed ○ Not addressed N/A Information not available

also excluded from the comparison. These represent acknowledged limitations of the current work and constitute open directions for future research, as discussed in Section 6.

The legend accompanying Table 1 clarifies the level of support each study provides for the evaluated characteristics. A property is *addressed* by the approach if it explicitly describes, proposes, or implements the corresponding characteristic. A property is *partially addressed* if it is considered or implemented to a limited extent. A property is *not addressed* if it is neither implemented nor discussed as part of the corresponding approach. Finally, the *information not available* label is used when insufficient details are provided on the property. This classification ensures a transparent, structured comparison, facilitating the identification of strengths, limitations, and research gaps across the reviewed studies. Consequently, the results reveal a clear gap in the application of agent-based monitoring, learning-based systems, proactive decision-making, and operations support in self-adaptive service-based systems.

Multiple similar approaches have been used for self-adaptive software development including adaptive service meshes [12, 13, 43, 48], gossip-based systems [3, 56], decentralized MAPE-K [6, 50, 65, 66], and multi-agent

systems [30, 46, 58, 61]. An adaptive service mesh is a self-managing communication layer for self-adaptive systems that provides traffic management, observability, and security [29]. It typically collects traffic patterns, detects security violations, security anomalies, and overloaded nodes, and performs self-adaptive actions such as rerouting, circuit breaking [18], scaling services, and enforcing security policies. Both service colonies and service mesh respond to runtime signals/messages and to policy-driven behaviors; for example, incoming message rates are used for decision-making. However, service mesh primarily adapts network or data-layer behavior rather than application decomposition.

Gossip-based systems use gossip (epidemic) protocols for data dissemination across nodes in the system [34, 56]. Gossip-protocol-based self-adaptive systems monitor data disseminated via the gossip protocol, analyze it, and make adaptive decisions [3]. Typically, such systems trigger routing changes, resource scaling, workload rebalancing, and data replication. Both service colonies and gossip-based systems avoid centralized control and decision-making, relying on local interactions. The purpose of a gossip-based system is efficient dissemination and failure detection to obtain eventual consistency [3]. At the same time, service colonies rely on explicit re-architecting decisions to ensure the target performance guarantees.

The predominant technique for developing self-adaptive software systems is the MAPE-K control loop [15], which organizes the system's main activities via a feedback cycle that starts with relevant data collected from the system environment. It monitors, analyzes, plans, and executes self-adaptive operations in the software systems [21, 40, 44]. SASSY [41], MUSIC [26], and REACT [53] are example frameworks for software systems with MAPE-K control loops in which all the data collection, analysis, and decision making are centralized. Adaptation decisions are primarily based on stakeholder-defined, scenario-based policies and pre-defined QoS models [41]. In contrast, decentralized MAPE-K [6, 50, 65] employs distributed peer loops in monitor, analyze, plan, and execute phases, and a knowledge model for self-adaptation across multiple domains (cloud, IoT, CPS) [50]. Both decentralized MAPE-K and service colonies are feedback-oriented, with continuous monitoring, analysis, and decision-making involving local knowledge and distributed loops. Moreover, existing patterns of regional planning and hierarchical control [66] align with service colonies. Service colonies are a prescriptive architectural style, whereas MAPE-K is a general control model used across domains. Furthermore, service colonies are based on proactive structural changes.

Multi Agent Systems (MASs) [30] share many similarities with service colonies, including autonomy, goal-directedness, and the ability to coordinate and negotiate. MAS is a broad engineering paradigm that employs autonomous, interacting agents for complex distributed problem-solving, including autonomous robotics, the smart grid, AI, and simulated systems, and widely uses game theory and cognitive science [30]. Service colonies are MASs. They integrate MAS principles into a microservices architecture, with concrete runtime operations like splitting, merging, and deployment-aware behavior. A service colony is a self-organizing group of services that adapts through decentralized decisions and emergent behavior.

Furthermore, multiple studies employing specific architectural models for self-adaptive systems have been reported in the literature. Rainbow [19] specifies an abstract architecture model to validate the system's runtime properties, evaluate them against a model of constraint violations, and perform global and module-level system adaptations. 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 [23, 24]. In this approach, a system is seen as composed of multiple constituent systems. Each constituent includes a manager that keeps track of system metrics directly related to the overall system's quality-of-service objectives. Each manager selects actions to improve the system's performance. The approach employs a hierarchical control strategy that applies control theory to regulate system behavior while outsourcing decision-making responsibility to individual meta-manager units. The self-adaptive, requirement-driven approach for auto-scaling microservices [49] focuses on efficiently allocating resources based on given requirements. Furthermore, auto-scaling is a highly specialized area in microservices and containerized

application optimizations that optimize CPU and memory usage by introducing auto-scaling algorithms [28, 62]. These studies do not provide architectural adaptations.

Finally, state-of-the-art studies on self-adaptive systems focus on cloud-based services, such as IoT and IaaS, rather than service-based software systems [67]. These approaches, which span across various domains (robotic, IoT, communication, automotive, e-commerce), use centralized monitoring, pre-defined models, and rule-based techniques [5, 35, 67]. Filho et al.'s systematic mapping study [17] showed that state-of-the-art self-adaptive microservices-based systems focus on infrastructure layer adaptations [32, 39, 57] or multi-layer adaptations [22, 37, 68]. The introduction of self-managing and self-architecting principles in software systems could increase the sustainability of (micro)service-based architectures, motivating our own research.

3 Service Colonies

This section introduces the components and interactions that define service colonies. It provides a detailed discussion of the structural elements of inhabitants, focusing on their adaptive mechanisms and how these mechanisms support the dynamic evolution of the colony. Furthermore, the service colony concept is framed as a self-adaptive software system.

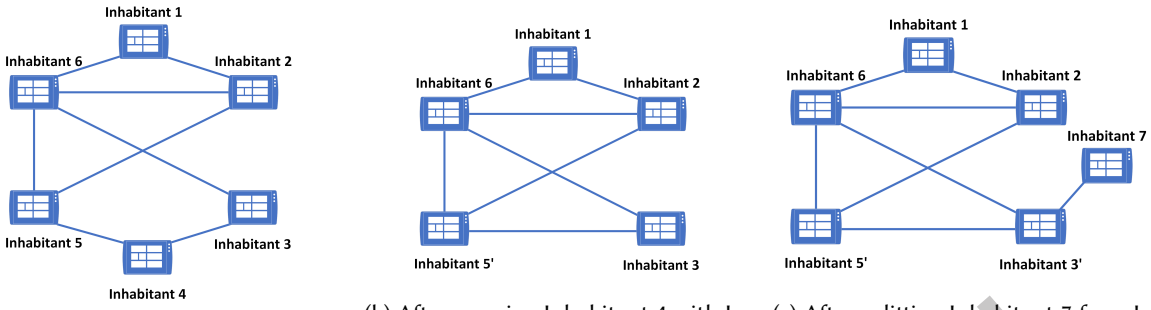
3.1 Components and Interactions

A *service colony* is a system composed of interacting software services, referred to as *inhabitants*, that collaborate to perform specific functions. These inhabitants are deployed in a distributed computing environment, such as a cloud platform, edge computing infrastructure, or Internet-of-Things devices, enabling the system to operate across diverse and decentralized resources. Depending on its design, a service colony can exhibit varying degrees of decentralization in its control and decision-making processes. These range from fully decentralized configurations, where individual inhabitants make autonomous decisions, to divisional or hierarchical structures, and even fully centralized models. This flexibility allows service colonies to adapt to different operational contexts, balancing scalability, responsiveness, and the complexity of coordination.

An *inhabitant* of a colony is an autonomous entity responsible for performing a specific functionality, or *service*, within the system. Each inhabitant has distinct roles and capabilities, reflecting its responsibilities and goals within the colony. Inhabitants aim to optimize their individual performance while contributing to the overall system's efficiency and quality of service. As self- and situationally-aware entities, they continuously monitor their performance relative to their assigned responsibilities, service obligations, and environmental conditions. Based on this awareness, inhabitants plan and execute actions to maintain or improve service quality. These actions may include replicating themselves, migrating to a more performant compute node, splitting into smaller entities, or integrating with inhabitants from another colony. Additionally, inhabitants can leverage learning mechanisms, such as reinforcement learning or rule-based adaptation, to refine their operations and decision-making based on prior experiences, enabling continuous improvement and adaptability.

An inhabitant interacts with other inhabitants and the environment by requesting services, providing services, or engaging in collaborative activities. Interaction patterns in a service colony can range from simple, direct message exchanges between two inhabitants to complex collaboration protocols involving multiple inhabitants and environmental inputs. These protocols may include synchronous or asynchronous communication, negotiation mechanisms, or coordinated workflows. Through these interactions, inhabitants coordinate their activities, exchange information, and interface with the system's users by processing inputs and delivering outputs, ensuring seamless operation and functionality within the colony.

The environment of a service colony comprises its inhabitants, the computing infrastructure supporting the colony, and the communication channels that enable interactions among them. Special components of the environment are the users, who interact with the colony to accomplish tasks and functions. Users achieve



(a) An example service colony.

(b) After merging Inhabitant 4 with Inhabitant 5.

(c) After splitting Inhabitant 7 from Inhabitant 3.

Fig. 2. Service colony adaptation process.

their goals by engaging with the colony's interface entities and receiving the results of their requests. This environment is inherently dynamic, as the state of its components and interactions evolves, and stochastic, due to the unpredictable nature of user requests, network conditions, and resource availability.

The service colony has decentralized monitoring, with inhabitants acting as autonomous agents. In general, there is no special inhabitant responsible for overall communication, monitoring, or dynamic adaptations. Instead, each inhabitant can learn from its behavior and automatically initiate actions to adjust the system. Each individual in a service colony can be (sub-)optimal, but collectively they aim to optimize the overall system performance. Figure 2a shows 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 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 changes 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.

3.2 Inhabitants

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 with 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. 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 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 through its behavior and to optimize the individual and overall system objectives via adaptive learning.

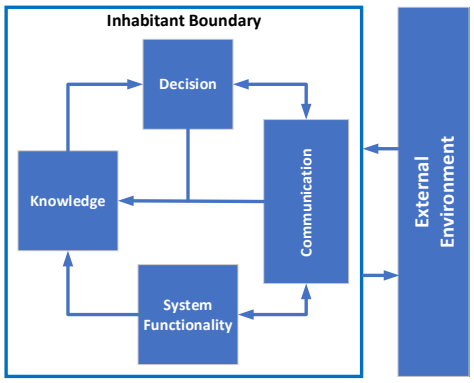


Fig. 3. An example architecture of a service colony inhabitant.

Figure 3 depicts an example 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 through the established communication links. 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 functions distributed among 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 other 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 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 data origin 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 by 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, using behavioral messages. Second, it analyzes inhabitant and environmental data to identify system operations, such as those relevant to rearchitecting the system, and actions to improve 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 Adaptations

An inhabitant can replicate itself. For example, if an inhabitant experiences a high volume of requests and no further optimizations are possible, it can decide to replicate itself to accommodate the increased workload.

A service colony can adapt its structure to the environment and optimize its communication patterns. Four basic operations are needed to support such adaptations: merging 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 *merge* 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 resource utilization in the system, thereby reducing operational costs. A complex merge 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 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 can add or remove communication links in the colony.

Figure 2b depicts the re-architectured system from Figure 2a after merging Inhabitant 4 with Inhabitant 5. This operation leads to redirecting the communication links of Inhabitant 4 to and from the resulting Inhabitant 5′. An example splitting of Inhabitant 3 is depicted in Figure 2c. In this case, Inhabitant 7 is split out of the Inhabitant 3 in Figure 2b, creating Inhabitant 7 and Inhabitant 3′. Hence, a communication link is established between the resulting inhabitants. Inhabitants can use this link to request services from one another.

One can develop different strategies for deciding which adaptations to the system structure to implement. Such strategies can result in the authoritative implementation of the intended adaptation or an adaptation confirmed by peers, a group of inhabitants, or 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. For example, this communication can follow an approach similar to the two-phase commit protocol [10]. If the change affects a group of colony inhabitants, permissions for the change can be requested from the affected inhabitants.

For instance, Inhabitant 5 in Figure 2a decides to merge with Inhabitant 4, and must obtain permission from Inhabitant 3 since Inhabitant 4 has a direct communication link with Inhabitant 3. The execution of an adaptation decision is initiated by sending a message to the relevant inhabitants requesting 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 change that was executed. These updates can involve adding, removing, or updating links between inhabitants.

Figure 4 illustrates the scenario of merging two inhabitants, 4 and 5. In this case, Inhabitant 5 identifies the need to merge and acts as the leading inhabitant of the change process. First, it seeks to get confirmation from Inhabitant 4, the inhabitant it intends to merge with. Once Inhabitant 4 accepts the merging request, the next confirmation request is sent to Inhabitant 3 to get approval to merge. If the inhabitants respond positively, the merging process commences. After the merge process completes, the execution confirmation message is sent to Inhabitant 3 to update relevant configurations. Figure 5 illustrates the sample communication sequence for splitting Inhabitant 3 illustrated in Figure 2c. Inhabitant 3 requests to split from Inhabitant 5′ and Inhabitant 6. Upon the acceptance of both requests, the splitting task is confirmed, and the splitting process is executed. After successful splitting, a confirmation is sent to the inhabitants. 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.

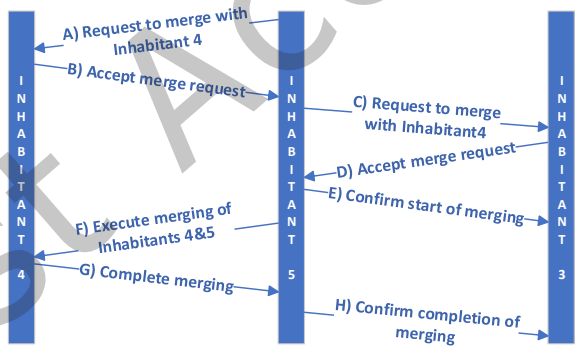


Fig. 4. Agreement for merging Inhabitants 4 and 5.

4 Implementation and Proof of Concept

A "Todo" item management application has been developed as a proof of concept prototype to validate the effectiveness of service colonies.[3] The prototype supports basic splitting and merging operations of inhabitants.

[3]https://github.com/thakshilad/ServiceColonyResources

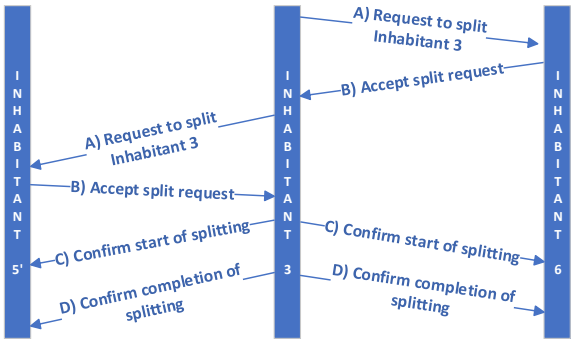


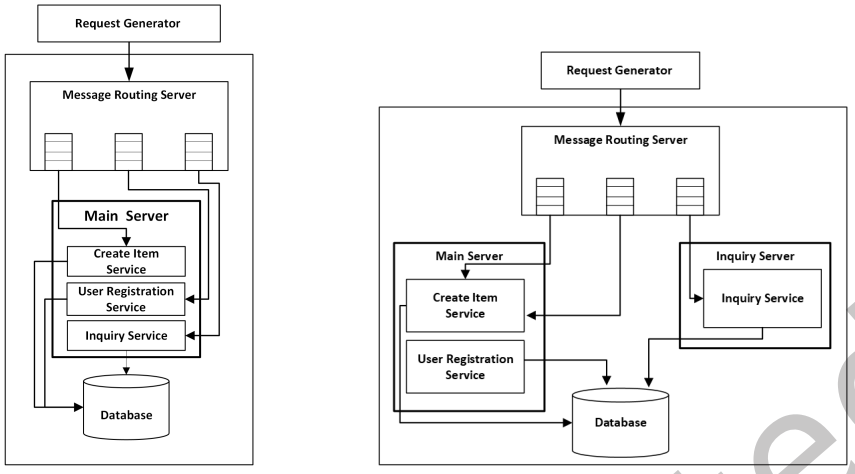
Fig. 5. Agreement for splitting between Inhabitants 3, 5′, and 6.

While a service colony, in general, is envisaged to support proactive decision-making by its inhabitants, for simplicity, the prototype implements rule-based, reactive decision-making. A service colony can identify services requiring splitting through comprehensive system data analysis. Once a split is confirmed, the service will be automatically deployed on the appropriate server(s). During startup, the service would communicate with other colony services, update the colony configurations, and begin handling service requests. To test the effectiveness of the service colony concept, pre-deployed services (split and merged) were used instead of automated extraction and deployment.

The Todo item management application consists of three services: Create Todo Item, User Registration, and Inquiry. The Create Todo Item service creates and persists new todo entries, the User Registration service creates and persists system users, and the Inquiry service performs a database search to retrieve a given todo item. A message-routing service directs incoming requests to the appropriate service, maintaining a dedicated queue of requests for each service. A separate tool, Request Generator, was developed to simulate user behavior by concurrently sending messages to each service at controllable rates. During testing, the Create Todo Item service and the User Registration service were always deployed as a single inhabitant of the colony on the main server. The Inquiry service was designed to support merging with or splitting from this single inhabitant. Initially, all three services were deployed on the main server as a single inhabitant, as shown in Figure 6a. This illustrates a scenario in which an inhabitant implements multiple services. If this inhabitant decides to split, the inquiry requests are handled by a dedicated inquiry inhabitant deployed on a separate server, as depicted in Figure 6b. If the two inhabitants decide to merge, the system re-architects itself back into the configuration shown in Figure 6a. The Inquiry service was chosen for split-off and merge-back operations because it involves querying data from the database (retrieving stored items), whereas the Create Todo Item and User Registration services perform only direct, lightweight database insertions. The split and merge operations were simulated by routing messages between the main and inquiry servers, both of which were pre-deployed to test the service colony operations.

The following metrics were used to model and evaluate the adaptive behavior and performance of the service colony:

- **Response Time (RT)** is the time it takes from when a user or request generator issues a request to the system until it receives the corresponding response. The response time is measured for each request to the Create Todo Item, User Registration, and Inquiry services.



(a) Merged system. (b) Splitted system.

Fig. 6. Two service colony deployments.

- **Average Response Time (ART)** reflects the overall user-perceived latency of the system measured as $1/N \sum_{i=1}^{N} RT_i$ where N is the total number of requests.
- **Adaptation Effectiveness (AE)** measures the gain in average response time after restructuring the system as follows:

$$AE = \frac{ART_{\text{before}} - ART_{\text{after}}}{ART_{\text{before}}},$$

where:

$$ART_{\text{before}} = \text{Average response time in the interval before adaptation, and}$$
$$ART_{\text{after}} = \text{Average response time in the steady state after adaptation.}$$

The adaptation effectiveness values are interpreted as follows:
  - $AE > 0$: effective adaptation—the restructuring reduces the average response time,
  - $AE = 0$: no impact—the restructuring does not affect the average response time, and
  - $AE < 0$: ineffective adaptation—the restructuring increases the average response time.
- **Throughput (TP)** is defined as N/T_{total}, where N is the number of successfully processed requests and T_{total} is the total observation time.
- **Queue Size (QS)** is the number of pending requests waiting in the service-specific request queue at time t. Adaptive decisions for merging and splitting use the queue size as a parameter.
- **Incoming Request Rate (IRR)** is defined as the number of requests arriving at a given service per unit time. This measures the external load imposed on a service and serves as a property for adaptation decisions. It is defined as $IRR(t) = N(t, t + \Delta t)/\Delta t$, where $N(t, t + \Delta t)$ denotes the number of requests arriving during the time interval $[t, t + \Delta t]$, and Δt represents the sampling interval (set to one second in the experiments conducted in this study).
- **Memory Consumption (MC)** represents the amount of heap memory actively utilized by the Java Virtual Machine (JVM) at time t. It is defined as $T(t) - F(t)$, where $T(t)$ denotes the total heap memory and $F(t)$ denotes the free heap memory at time t. Both quantities are obtained using the Java Runtime API.

- **Memory Gain (MG)** quantifies the relative reduction in total memory footprint achieved due to restructuring by the service colony compared to the baseline over the execution time window. Time weighted mean memory usage, $\bar{M}$ for the duration T calculated as:

$$\bar{M} = \frac{1}{T}\int M(t)\,\mathrm{d}t\,.$$

Memory Gain (MG) is calculated as:

$$MG = 1 - \frac{\bar{M}_s}{\bar{M}_b}\,,$$

where $\bar{M}_s$ and $\bar{M}_b$ are the time weighted means of service colony and baseline/initial systems, respectively. The values are interpreted as follows:

- $MG > 0$: Effective utilization—reduced memory usage compared to the baseline.
- $MG = 0$: No impact—no effect on memory usage relative to the baseline.
- $MG < 0$: Ineffective utilization—increases memory usage compared to the baseline.

4.1 Self-Adaptation Description Language for Service Colony

Modeling self-adaptive systems is inherently challenging due to uncertainty in their operational environments. Self-adaptive capabilities further increase this difficulty because adaptation-related features are distributed across multiple parts of the system. As a result, models often become tightly interwoven, making the system harder to understand [38].

To address this complexity, several Domain-Specific Modeling Languages (DSMLs) have been proposed [16]. The Adapt Case Modeling Language (ACML) is one such DSML. It follows the Concern-Specific Modeling Language (CSML) approach, which is built on top of the Unified Modeling Language (UML). ACML separates concerns explicitly and provides dedicated constructs for specifying self-adaptive behavior [38]. Therefore, ACML is adopted as the description language for service colonies.

ACML comprises two major components: the adaptation view, which is the structural model, and adapt cases, which are behavioral models of the system. The adaptation view illustrates core components of the system with sensors, effectors, adaptation properties, and knowledge, as a UML-like component view. A sensor is an interface/class that enables the system to observe or measure relevant properties. An effector is an interface/class used by the system to perform actions that change its state or the environment.

Figure 7 shows the sample low-level adaptation view of the service colonies, including sensors and effectors. It contains three inhabitants, X, Y, and Z. The internal module structure is depicted only for the inhabitant X. Both inhabitant Y and Z have a similar structure. Thus, for simplicity, only their interaction with the colony is depicted in the figure.

The knowledge module is responsible for acquiring and maintaining information relevant to the colony and its inhabitants. To support this role, sensing and context-awareness capabilities are embedded in the module, enabling it to maintain an up-to-date view of both the inhabitant's state and its operating environment. The knowledge module maintains several categories of context information, including the runtime status of the colony, the current topology of interconnected inhabitants, adaptation policies, Service Level Agreements (SLAs), and adaptation history. This contextual knowledge is derived from data captured through a set of sensors embedded in the knowledge module. Service colonies incorporate multiple sensor types to observe both operational and structural conditions. The most fundamental sensor is the service load sensor, which monitors the incoming request rate per inhabitant. Another key sensor is the performance sensor, which, in the implemented prototype, measures queue size, i.e., the number of pending requests awaiting processing by an inhabitant. In addition, each inhabitant continuously monitors its own operational status and contributes information about overall resource

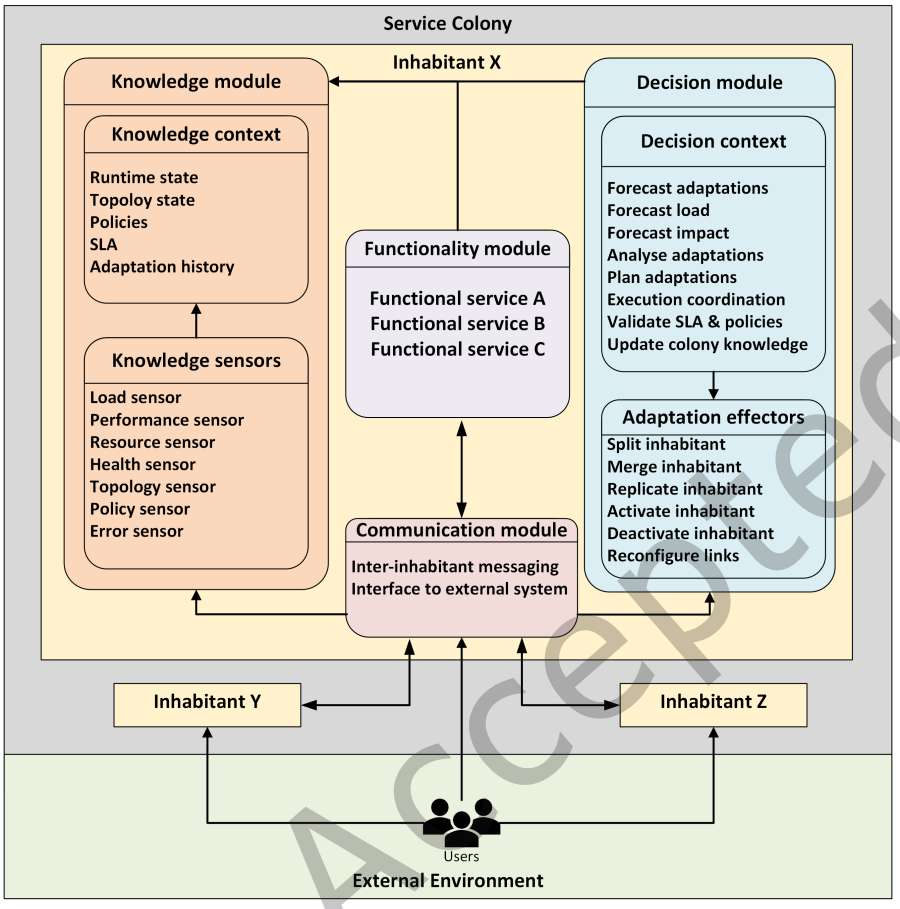


Fig. 7. ACML adaptation view: An example model for service colonies.

utilization within the colony, thereby acting as a distributed sensing entity. Moreover, the knowledge module captures the changes in colony topology, policy updates, and inhabitant-level errors or failures.

The decision module functions as the adaptation effector and control unit, performing reasoning and executing adaptive decisions. It is responsible for analyzing contextual information captured by the knowledge module and triggering appropriate adaptation actions when required. Figure 7 illustrates a sample set of actions performed by the decision module during runtime adaptation. The decision module performs a range of operations, including identifying the need for adaptation, analyzing and forecasting load variations, assessing the impact of potential changes, evaluating adaptation risk, planning the adaptation process, and requesting coordination or approval from relevant fellow inhabitants when required. In addition, it is responsible for executing adaptation decisions, validating against applicable SLAs and policies, and updating the knowledge module once the adaptation process has been completed. These controlling functions lead to the selection and execution of adaptive decisions, which are carried out through a set of adaptation effectors. The service colony architecture provides effectors to support key structural and behavioral adaptation actions, including splitting, merging, replicating, activating/deactivating inhabitants, and runtime reconfiguration of colony relationships and interactions.

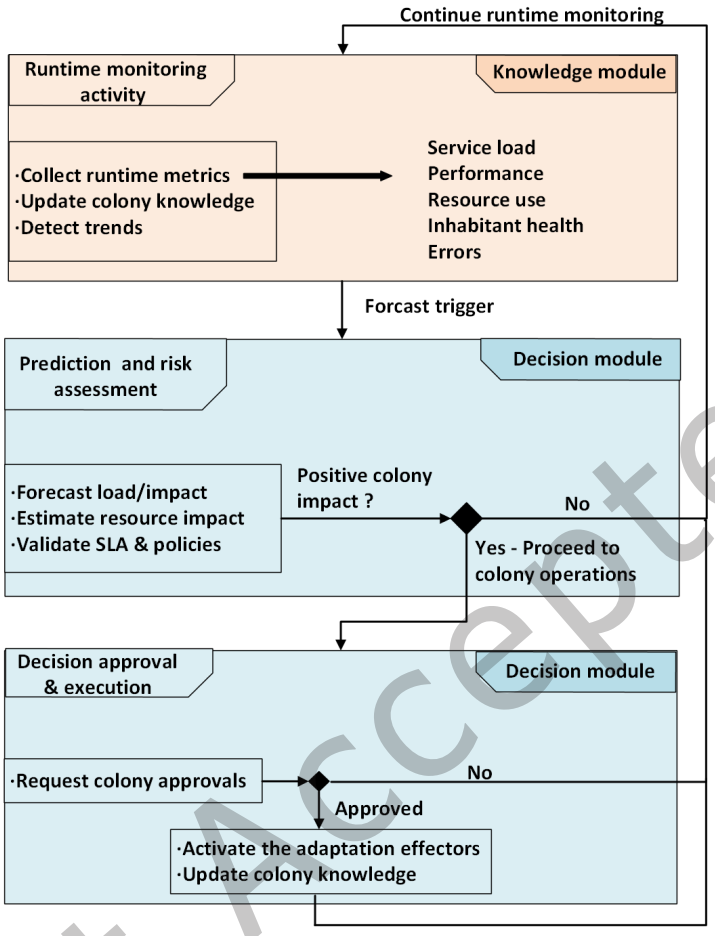


Fig. 8. Example ACML adaptation cases for service colonies.

The system functionality module is responsible for performing domain-specific business functionalities of the inhabitant. Through this module, an inhabitant can provide one or more services that are exposed to external users or other inhabitants within the colony.

The communication module manages the communication and message exchange between inhabitants and the external environment. It acts as the interaction layer through which functional service requests and inhabitant-related messages are transmitted. These may include business requests, adaptation-related notifications, and other coordination information required for colony operation.

The adapt cases of the ACML model define how the system monitors its environment and the type of actions it should perform in response. Examples of adapt cases are shown in Figure 8. It captures the logic for monitoring and knowledge capture, predicting possible adaptive operations, performing impact analysis, and executing adaptive operations via effectors.

4.2 Experimental Results

Four test scenarios were conducted to evaluate the performance of the service colony with respect to the defined metrics. The incoming request rate and queue size were selected as key parameters for adaptive decision-making due to their ease of measurement and real-time availability.

The testing setup contains five Ubuntu servers and one MySQL database instance. Each server has 4 VCPUs, 16GB RAM, and a 30GB hard drive. The database instance has 4GB RAM and a 1GB hard drive. The service colony logged its operations under different workloads, which were subsequently analyzed. The testing scenarios and the results are discussed below.

4.2.1 Scenario 1: Splitting.

This scenario evaluated the performance of the split function of the service colony. The following three cases are evaluated in this scenario:

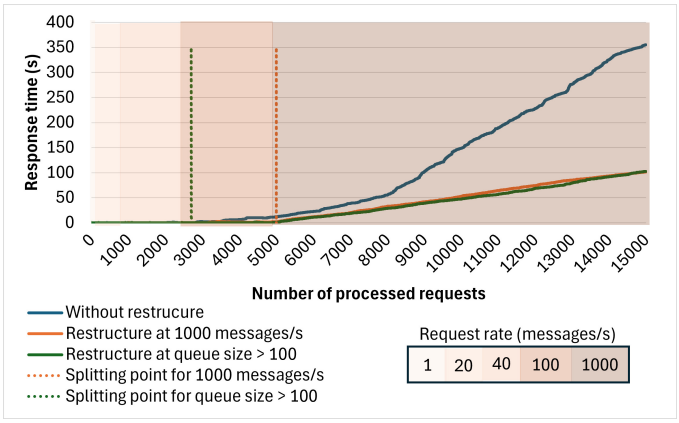
Case 1: No splitting. All the requests are served from the main server (Figure 6a).
Case 2: Splitting the Inquiry service at the IRR reaches or exceeds 1,000 requests per second (Figure 6b).
Case 3: Splitting the Inquiry service when the number of requests to be processed in the Inquiry requests queue size exceeds 100 (Figure 6b).

Case 1 represents the behavior of the original software system, while Cases 2 and 3 simulate an inhabitant splitting in a service colony under overload conditions.

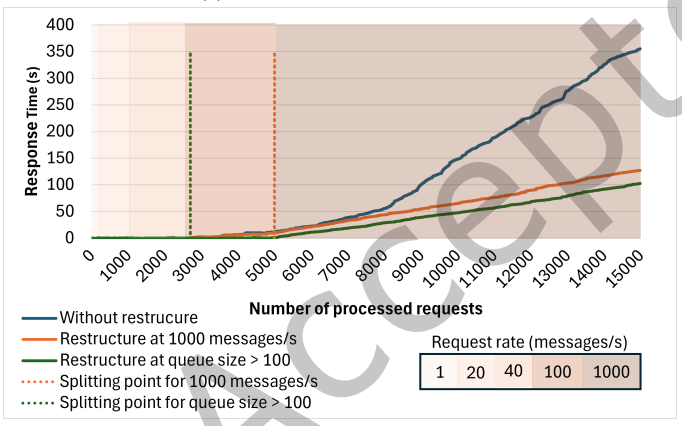
Initially, all three services were deployed on the main server, as depicted in Figure 6a. To simulate overload conditions, the request generator progressively increased the IRR from 1 request per second to 1,000 requests per second for each service for all the cases. The selected IRR range and corresponding thresholds were determined through empirical evaluation of the system. Within this interval, response time exhibited significant variation as the workload increased. Beyond 1,000 requests per second, the system reached a saturation state in which response time remained relatively stable despite further increases in IRR. Therefore, 1,000 requests per second was chosen as the upper bound.

All services were subject to gradually increasing workloads to simulate server overload conditions. The Inquiry service is latency-sensitive due to database lookups. Therefore, the service colony activates the splitting function for the Inquiry service when either IRR or QS exceeds a predefined threshold. In scenario 2, the IRR threshold was set to 1,000 requests per second, as this value corresponds to the saturation point of the system. In scenario 3, the QS threshold was set to 100, based on empirical observations indicating this value as the onset of response time degradation with increasing queue size. After the IRR reached 1,000 requests per second, the load was maintained to keep the system saturated and prevent the services from merging onto a single server during the experiment. Application response time was measured to evaluate the performance impact of the splitting function. In addition, memory consumption was analyzed to assess resource utilization.

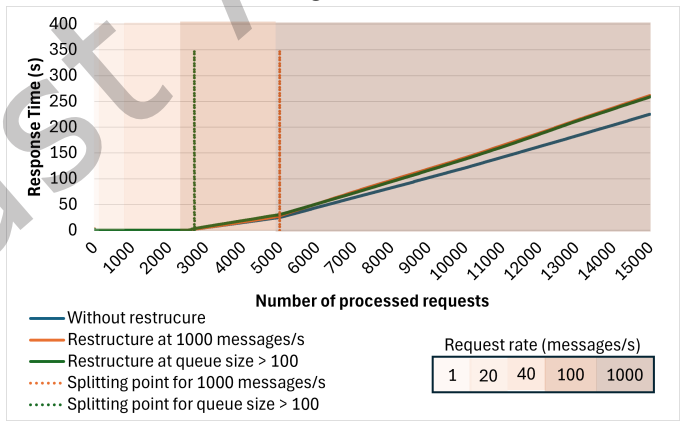
Figure 9 shows the response time analysis of the three services over time. While the Inquiry service does not display a significant improvement following the splitting, both the Create Todo Item and User Registration services experience a notable reduction in response time. This reduction is attributed to enhanced resource availability after offloading the resource-intensive inquiry request processing. Furthermore, splitting based on queue size marginally outperforms splitting based on response rate. Figure 10 depicts the total memory consumption for the three cases. As expected, the total memory consumption by the system is negatively impacted since splitting introduced an additional server to the system. In summary, the results reveal a seamless transition between the two architectural configurations, both before and after splitting, accompanied by a considerable improvement in response time.



(a) Create Todo Item service.

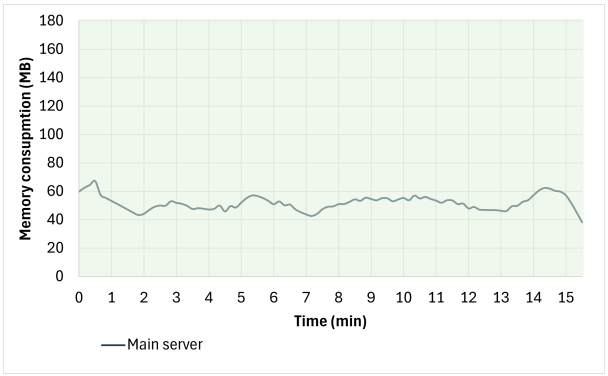


(b) User Registration service.

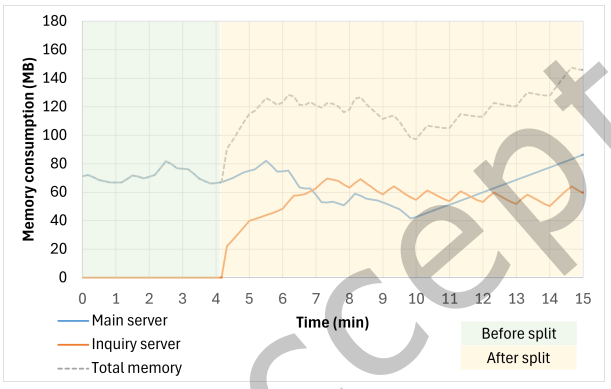


(c) Inquiry service.

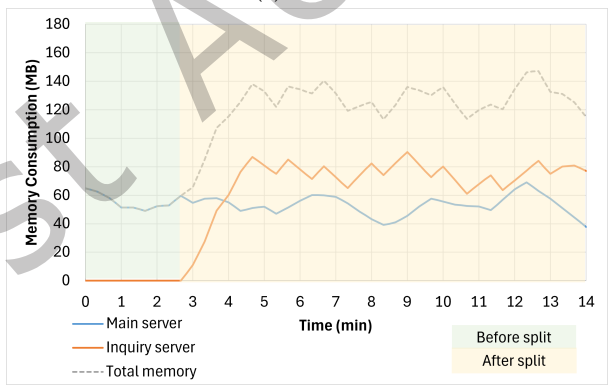
Fig. 9. Scenario 1: Response time analysis (splitting).



(a) Case 1.



(b) Case 2.



(c) Case 3.

Fig. 10. Scenario 1: Memory consumption analysis (splitting).

Furthermore, a statistical analysis of the results was conducted based on the defined metrics: throughput, average response time (ART), adaptation efficiency (AE), and memory gain (MG), and the results are depicted in Table 2. According to the metrics' definitions, higher throughput indicates better performance, whereas lower

ART reflects improved system responsiveness. Additionally, positive values of AE and MG represent performance improvements in the service colony compared to the baseline configuration. The comparative analysis confirms that, in the splitting scenario (Scenario 1), the service colony architecture improves throughput, reduces average response time, and yields positive adaptation efficiency for create-todo and register services. However, this configuration adversely affects memory utilization, and the split service's (Inquiry) performance may degrade.

4.2.2 Scenario 2: Merging.

This scenario tested the merging of inhabitants. Initially, the system operates in the deployment mode shown in Figure 6b. After observing a low inquiry request rate, it merges its two inhabitants into one as depicted in Figure 6a.

Throughout the experiment, the Create Todo Items and User Registration services maintained a constant IRR of 20 requests per second. This controlled workload ensured that the server hosting these two services was not overloaded when the inquiry service was merged back. A minimum stable workload was required to preserve consistent system behavior. Based on the empirical evaluation, an IRR of 20 requests per second provided stable performance. Therefore, this rate was maintained for both services during the experiment.

In contrast, the Inquiry service was used to simulate varying load conditions. Its request rate was gradually reduced to emulate a decreasing IRR, starting from 1,000 requests per second and decreasing to 1 request per second. An IRR of 1,000 requests per second was sufficient to overload the system. Accordingly, the Inquiry service simulated a transition from high-load to low-load conditions by decrementing the IRR from 1,000 to 1 request per second. The queue size was not used in the merging decision in this experiment to avoid responding to temporary queue fluctuations caused by decreasing IRR values. We analyze two cases in this scenario:

Case 1: No merging. The main server runs one inhabitant that contains the Create Todo Item and User Registration services. The Inquiry service is deployed on a dedicated server (Figure 6b).

Case 2: The Inquiry service merges with the Create Todo Item and User Registration services and is deployed on the main server (Figure 6a) when the request rate decreases to one message per second.

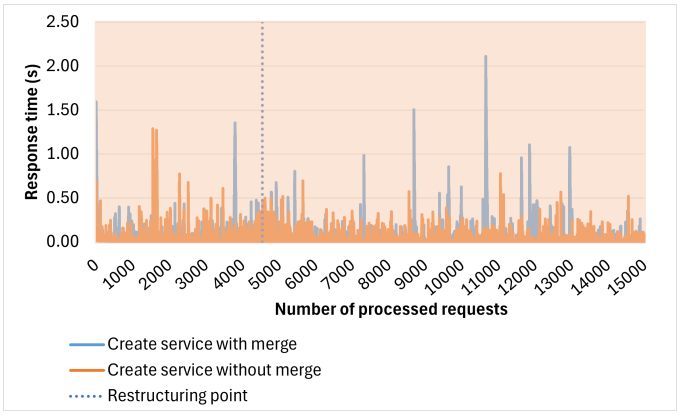
Case 1 simulates a software system with two services that do not support merging operations, while Case 2 validates merging functionality within the service colony. This scenario analyzes both response time and resource utilization, comparing the performance of a system employing the service colony with one that does not, thereby assessing the effectiveness of the merging function.

Figure 11 presents the results of the response time analysis. No significant impact on response time was observed in this scenario. However, a substantial reduction in total memory utilization was noted, as illustrated in Figure 12. The discontinuation of the inquiry server following the merging operation led to a decrease in overall memory consumption by approximately 50%, as illustrated in Figure 12b. Furthermore, Table 2 shows positive memory gains but negatively affects throughput, average response time, and adaptation efficiency relative to the baseline service. Therefore, this experimental scenario demonstrates that resource utilization can be reduced without adversely affecting system performance.

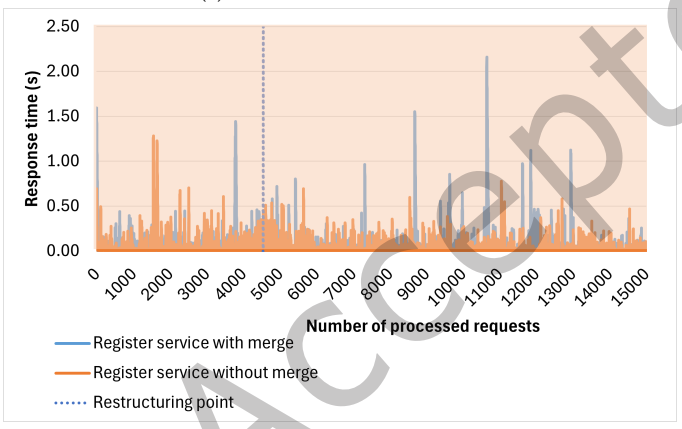
4.2.3 Scenario 3: Multiple Splitting and Merging.

In this scenario, we assessed the system performance under varying inquiry request rates to simulate the dynamic runtime operation of a service colony. As a result of these fluctuations, multiple merge and split adaptations occur during system operation.

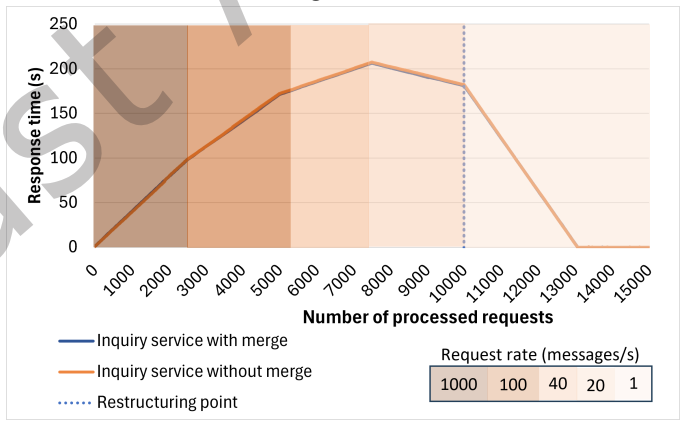
As in Scenario 2, the Create Todo Items and User Registration services were maintained at a constant IRR of 20 requests per second to ensure stable behavior. In contrast, the Inquiry service IRR follows a wave-like pattern to emulate real-world workload variations. Specifically, the rate begins at 2 requests per second, gradually increases



(a) Create Todo Item service.

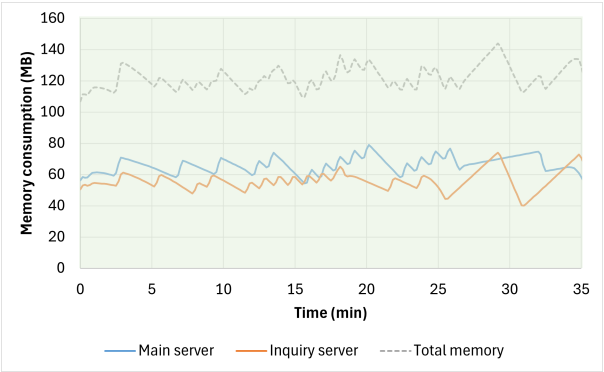


(b) User Registration service.

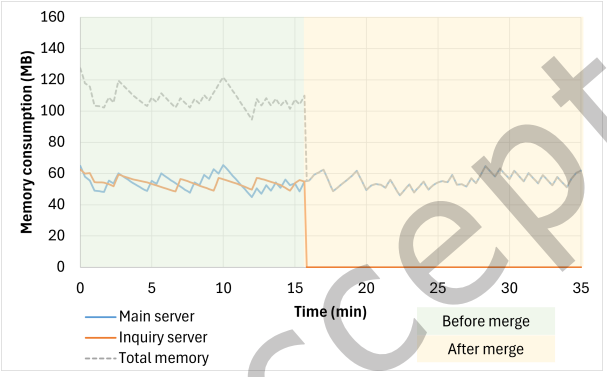


(c) Inquiry service.

Fig. 11. Scenario 2: Response time analysis (merging).



(a) Case 1.



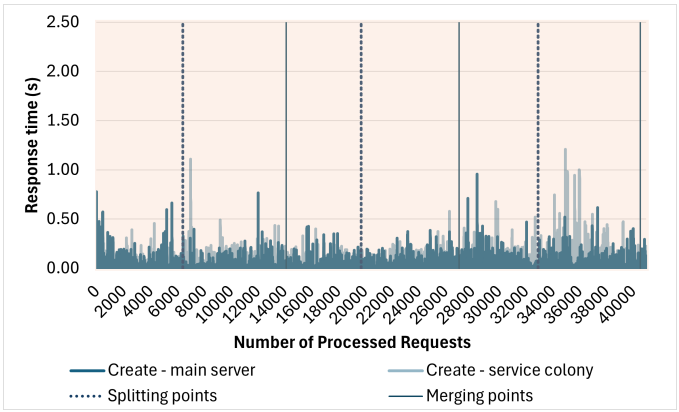
(b) Case 2.

Fig. 12. Scenario 2: Memory consumption analysis (merging).

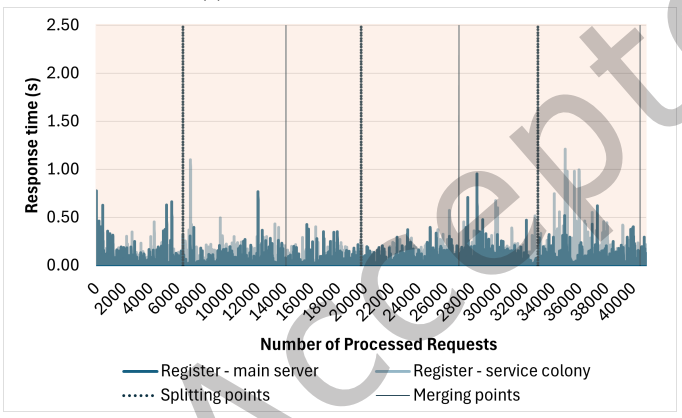
Table 2. Summary of results.

Scenario	Case	Throughput			ART			AE			MG
		Todo	Inquiry	Register	Todo	Inquiry	Register	Todo	Inquiry	Register	
1	1	0.021	0.039	0.021	307870.760	83832.268	309059.331				
	2	0.053	0.024	0.053	45662.401	185200.552	45722.105	0.851	-1.209	0.852	-0.997
	3	0.042	0.036	0.042	77873.044	96548.193	78243.879	0.747	-0.152	0.747	-1.155
2	1	0.019	0.020	0.019	35.614	112308.498	36.336				
	2	0.019	0.019	0.019	152.200	132198.792	155.248	-2.763	0.825	-2.774	0.352
3	1	0.019	0.007	0.019	16.791	141326.264	16.798				
	2	0.019	0.008	0.019	19.865	114015.614	20.028	-0.183	0.193	-0.192	-0.119
4	1	0.029	0.026	0.028	220557.237	273584.176	219514.829				
	2	0.052	0.029	0.052	141223.403	244031.893	140818.619	0.359	0.108	0.358	0.256

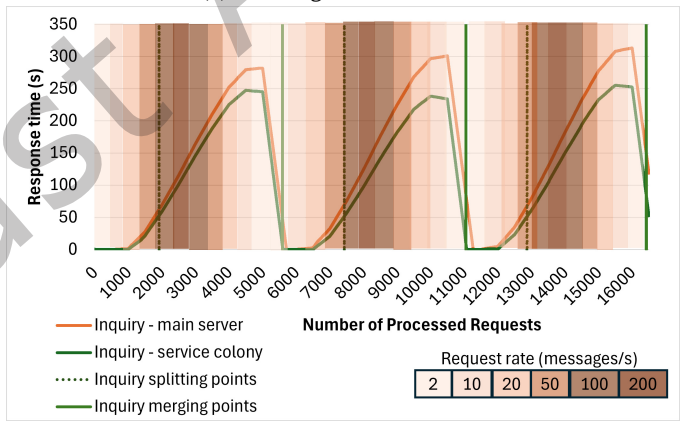
to 200 requests per second, then progressively decreases back to 2 requests per second, after which the cycle repeats. Although empirical evaluation indicates that the system becomes overloaded at an IRR of 1,000 requests



(a) Create Todo Item service.



(b) User Registration service.



(c) Inquiry service.

Fig. 13. Scenario 3: Response time analysis (splitting and merging).

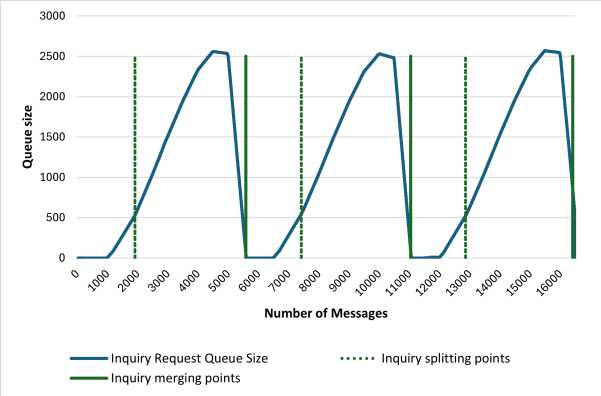


Fig. 14. Scenario 3: Inquiry queue size fluctuation

per second, the load variation range in this experiment was limited to 2-200 requests per second to reflect the typical operating conditions rather than extreme cases.

The split adaptation dividing the inhabitant executing all three services into two separate services, as illustrated in Figure 6b was triggered when the queue size exceeded 500. Conversely, a merge adaptation consolidating the system into a single server, as shown in Figure 6a was triggered when the inquiry queue size dropped below 50 requests. Since the request rate in this scenario was moderate, the queue size was selected as the primary adaptation metric, as it directly reflects system congestion. The threshold values were determined based on experimental observations of queue size fluctuations (see Figure 14). Specifically, the split threshold of 500 requests was chosen because queue growth becomes sharp beyond this point, indicating rapid load escalation. The merge threshold of 50 requests was selected because the request rate effectively declined to near zero below this level. During the experiment, merging and splitting were observed three times each. Response time and memory consumption were analyzed in these two cases:

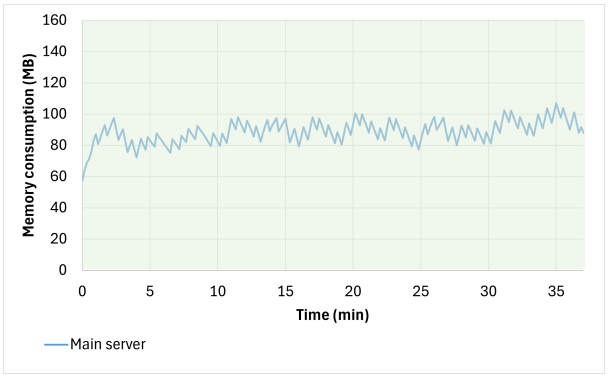
Case 1: The system is composed of a single inhabitant that implements all three services and executes on the main server without performing merge and split adaptations (Figure 6a).

Case 2: The system alternates between one and two inhabitant configurations shown in Figure 6a and Figure 6b, respectively, as required, according to the described adaptation rules.

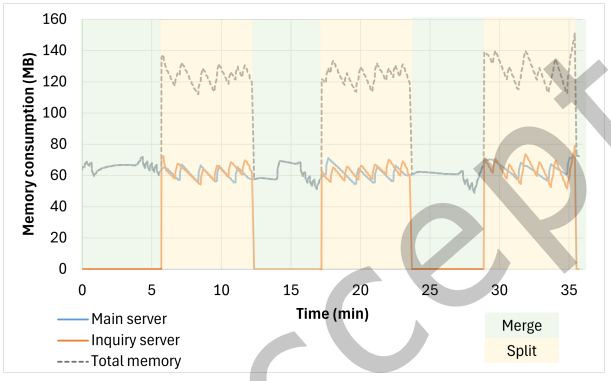
In this scenario, Case 1 represents the original software system, while Case 2 illustrates the behavior of the service colony system. The response times and memory consumption were analyzed and are presented in Figure 13 and Figure 15, respectively. Although the Create Todo Item and User Registration services are not significantly affected, the Inquiry service shows an improvement in response time, as demonstrated in Figure 13c. Furthermore, the improvement in inquiry response time becomes more pronounced with each re-architecture cycle. The average total memory consumption of the two-inhabitant system, shown in Figure 15b, is comparable to that of the single-inhabitant system deployed on the main server, as depicted in Figure 15a. Therefore, this scenario highlights that, during the continuous operation of a service colony involving multiple merging and splitting operations, response time can improve while maintaining a similar overall average resource consumption.

4.2.4 Scenario 4: Limited Resources.

Scenarios 1 to 3 assume unlimited resources. In many real-world scenarios, resources are costly and, thus, constrained. In Scenario 4, system performance was evaluated under limited resource conditions. The main server



(a) Case 1.



(b) Case 2.

Fig. 15. Scenario 3: Memory consumption analysis (splitting and merging).

CPU was stressed using the Ubuntu stress tool to simulate resource contention for the system. In addition, each inhabitant of the Todo application, implemented as a service colony, was started by limiting the Java Virtual Machine (JVM) memory allocation. Furthermore, memory constraints were simulated by allocating memory to arrays with 80 million entries. The response time of the three services and the memory consumption by the system's inhabitants were analyzed by allocating 32MB of memory per service. The memory allocation was chosen as 32 MB due to the JVM memory requirements for starting the system. Hence, during the execution, a total memory of 96MB was maintained across the system. These configurations were imposed to simulate the limited CPU and memory, thus simulating resource-constrained environments such as IoT and edge devices.

In this scenario, the service colony system was evaluated both with and without the splitting adaptation. Initially, the system was deployed with a single instance on the main server, hosting all three services within a single JVM, with appropriate memory allocations. Since JVM does not support dynamic memory allocation, upon reaching the splitting condition, the two inhabitants, one that implements the Create Todo Item and User Registration services, and the other that implements the Inquiry service, were deployed on fresh servers with the respective JVM memory allocations that maintain the necessary memory levels across the system. During the experiment, a constant IRR of 500 requests per second was applied to each service. This rate was selected based on empirical evaluations conducted under resource-constrained conditions, where an IRR of 500 requests

per second was observed to produce a saturated system state. The splitting adaptation was triggered when the inquiry request queue size exceeded 5,000 requests. This threshold is higher than in other test scenarios because the system operated under resource constraints, leading to longer response times and consequently faster queue growth.

As illustrated in Figures 16a and 16b, in the resource-constrained environments, the Create Todo Item and User Registration services exhibit better response time when the system is split into two inhabitants. When split, the response time of these two services increases linearly as more requests arrive, while it slows down substantially when a single inhabitant supports all three services. In turn, the response time of the Inquiry service shows a minor improvement when the system splits into two inhabitants, as depicted in Figure 16c. The processing of an inquiry request, on average, is more resource-demanding than the processing of other services. Hence, when three services are implemented as single inhabitant and are executed by single JVM, most of the allocated memory is eventually consumed by processing the inquiry requests, decreasing the overall performance of the system. When the Inquiry service is deployed on a separate JVM, the processing of the Create Todo Item and User Registration services accelerates substantially. In general, the system performs better when more memory is allocated. However, when the number of requests is overwhelming, this advantage gradually diminishes.

The memory consumption for the scenario is presented in Figure 17. Case 1 represents the original software system, while Case 2 illustrates the behavior of the service colony system in a resource-constrained environment. Figure 17a illustrates the memory utilization of the main server. The main server is almost saturated and utilizes approximately 90MB of the total memory allocation, which is 96MB. In contrast, the implementation of service colonies exhibits lower overall memory consumption despite the system being distributed across two servers, as depicted in Figure 17b.

Furthermore, Table 2 confirms that under resource-constrained conditions (Scenario 4), the service colony architecture exhibits the most favorable overall performance, achieving higher throughput, lower average response time, positive adaptation efficiency, and memory gain. These findings indicate that the service colony architectural style is particularly well-suited for resource-constrained environments.

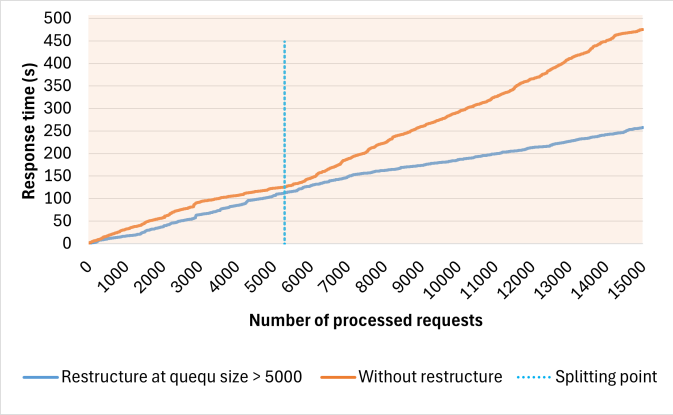
5 Benefits and Challenges

This section examines the benefits of implementing software systems as service colonies, as well as the challenges that may arise from such implementations.

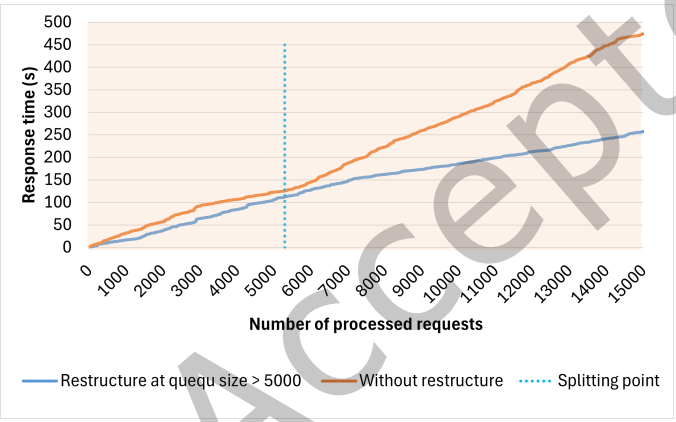
5.1 Benefits

Both service-oriented and microservices architectures enable the identification of loosely coupled services that can be deployed independently. These services can be designed, developed, and tested autonomously. Additionally, containerization techniques allow these services to auto-scale in cloud-based environments based on performance demands. Generally, self-adapting systems address specific domain requirements, resource optimizations, and infrastructure adaptations by employing reactive error-handling functionality in pre-modeled scenarios [35, 67]. In contrast, service colonies are not constrained by pre-identified or pre-modeled scenarios. Instead, a colony can proactively re-architect itself in a bottom-up manner through decentralized system analysis.

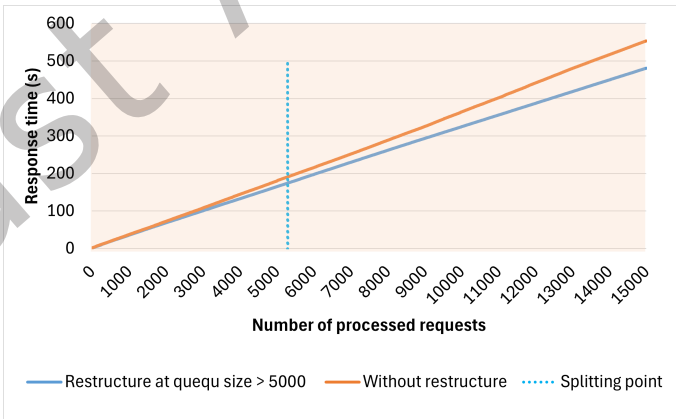
Consider an online shopping system developed as a service-based application. Assume that user registration and shopping cart actions are developed as two separate services within the system. In December, due to the festive season, the system receives a 100% surge in the volume of requests. During this 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



(a) Create Todo Item service.

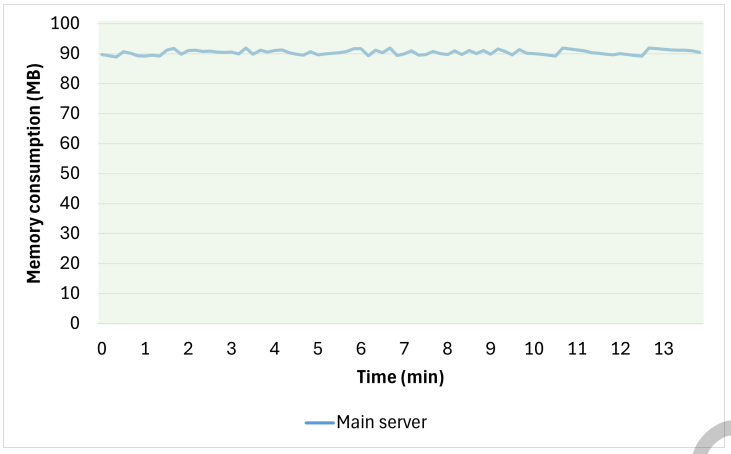


(b) User Registration service.

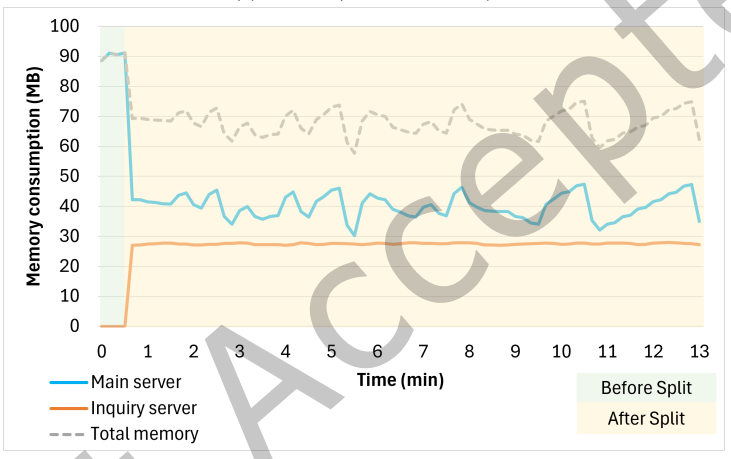


(c) Inquiry service.

Fig. 16. Scenario 4: Response time analysis (limited resources).



(a) Case 1 (one inhabitant)



(b) Case 2 (one inhabitant splits into two)

Fig. 17. Scenario 4: Memory consumption (limited resources).

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 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.

In general, we identify the benefits of a service colony to be at least the following:

- **Extension of microservices** Service colonies extend the capabilities of microservice-based architectures by inheriting their flexibility, resilience, modularity, optimized resource usage, and reduced operational overhead. As the industry increasingly adopts microservice-based systems, service colonies align with this trend, offering innovative 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. For example, in the proof-of-concept Scenario 1, each inhabitant independently monitors its own request rate and queue size, predicts overload conditions, and initiates adaptation decisions accordingly, without depending on a central monitoring agent to detect performance degradation and determine appropriate adaptation actions, thereby enabling proactive fault tolerance.
- **Continuous optimization** Through continuous learning from operational data at the individual inhabitant level, and consequently the entire service colony, progressively evolves into a self-optimizing system. This process enables ongoing performance refinement, analogous to experimental scenario 3, in which inquiry response time continuously improves across successive adaptive cycles.
- **Goal orientation** Both the service colony as a whole and its individual inhabitants have objectives and strive to achieve them. Properties and threshold values at the System-wide and individual-inhabitant levels can be defined within the system, as in the proof of concept. Hence, the system's collaborative nature fosters iterative improvements toward shared goals.
- **Dynamic service 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. In particular, as demonstrated in Scenario 4 in the prototype, where the system operates in resource-constrained environments, the service colony can achieve positive adaptation efficiency and memory gain with improved throughput and lower average response time.
- **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.
  Consistent with the experimental scenarios, the splitting function operates independently because it represents either the full or a subset of the original technical stack and, therefore, introduces no additional dependencies. Similarly, the merging function can operate effectively within heterogeneous environments, provided that the deployment server supports the technical stack required by the merging service.
- **Scalability** By integrating dynamic splitting and merging, the service colony enables individual scalability based on performance metrics. It supports proactive decision-making and predictive scaling initiated by its inhabitants, reducing maintenance costs and ensuring that resource allocation remains aligned with actual system demands.

5.2 Challenges

The challenges associated with the design, implementation, and maintenance of service colonies are yet to be fully understood and studied but we note below the challenges we identified while implementing and validating our service colony prototype (Section 4):

- **System complexity** A service colony is a dynamic, complex system composed of interacting components. 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 test service colonies properly. Additionally, validating the correctness of system decisions before re-architecting is challenging due to the system's dynamic nature, which makes it difficult to predict outcomes and ensure stability in advance. Leveraging the expertise of system specialists, the training of the decision-making process through the integration of machine learning, evolutionary algorithms, and neural networks could enhance the verification and validation processes.
- **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 re-architecting.
- **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 re-engineering of the system. Dynamic repartitioning and redesigning of databases can lead to data replication, which may result in data inconsistencies. In contrast, inadequate data partitioning may lead to performance bottlenecks during periods of high system load, ultimately diminishing the overall performance of the service colony.
- **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.
- **Re-engineering** Existing systems that wish to benefit from the advantages of service colony architecture need to be re-engineered accordingly. A systematic process for re-engineering software systems into service colonies must be defined to facilitate the migration of legacy systems or existing service-based systems to this new architectural style.

6 Conclusions and Future Work

This paper presented the concept of a *service colony*, an architectural style for developing software 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 and proactive observation of the environment, 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 induce system-level behavior. Consequently, service colonies aim to inherit the benefits of complex system architectures, including resilience, robustness, adaptability, scalability, and distributed control. We validated these qualities in a publicly-available study in which we measured the performance of a service colony using a proof of concept. Furthermore, our experimental scenarios demonstrated that service colonies yield improved outcomes, regardless of the effort required for system restructuring. Furthermore, it is feasible to transition from one steady state to another without restarting the services while maintaining the system's performance.

The proof-of-concept presented in this study constitutes an initial step toward realizing the vision of service colonies. While it demonstrates the feasibility of the proposed approach, further empirical validation is required to substantiate the claims made in this paper. As a primary direction for future work, we plan to conduct comprehensive surveys with industry practitioners and develop in-depth real-world case studies. These efforts will enable us to evaluate the applicability, scalability, and practical relevance of service colonies in operational environments. In addition, we aim to integrate the proposed model into a real-world application enriched with more sophisticated learning and decision-making algorithms to assess its performance under realistic conditions. Moreover, a systematic evaluation of the service colony and state-of-the-art self-adaptive systems remains an important avenue for future research.

Another important research direction is to develop a comprehensive framework for automated service deployment to address the current reliance on pre-deployed services. Designing and evaluating techniques to automatically identify opportunities to split and merge services based on patterns of frequent versus infrequent execution in the system is another promising direction. Furthermore, assessing adaptation efficiency and correctness remains an important direction for future research.

Additionally, exploring methods to establish communication interfaces—both between the inhabitants of service colonies and between these inhabitants and system users—remains an important avenue for research. Furthermore, integrating data analytics approaches to enable proactive decision-making, alongside designing and evaluating various colony inhabitant architectures and their interaction principles, will be crucial for understanding their impact on the system's overall behavior. Through systematic experimentation and comparative analysis, we seek to identify architectural configurations that maximize adaptability, resilience, and performance.

Acknowledgements. This work is partially supported by Australian Research Council Discovery Project (DP220101516), "Embedding Enterprise Systems in IoT Fog Networks through Microservices."

References

[1] Charles Abrams and Roy W. Schulte. 2008. Service-oriented Architecture Overview and Guide to SOA Research. Gartner Research Document G00154463.

[2] Ouanes Aissaoui, Fadila Atil, and Abdelkrim Amirat. 2013. *Towards a Generic Reconfigurable Framework for Self-adaptation of Distributed Component-Based Application.* Springer, 399–408. doi:10.1007/978-3-319-00560-7_43

[3] Emad Albassam, Jason Porter, Hassan Gomaa, and Daniel A. Menascé. 2017. DARE: A Distributed Adaptation and Failure Recovery Framework for Software Systems. In *International Conference on Autonomic Computing (ICAC)*. IEEE, 203–208. doi:10.1109/ICAC.2017.12

[4] Françoise André, Erwan Daubert, and Guillaume Gauvrit. 2010. Towards a Generic Context-Aware Framework for Self-Adaptation of Service-Oriented Architectures. In *International Conference on Internet and Web Applications and Services (ICIW)*. IEEE, 309––314. doi:10.1109/ICIW.2010.52

[5] Hugo Araujo, Mohammad Reza Mousavi, and Mahsa Varshosaz. 2023. Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review. *ACM Transactions on Software Engineering and Methodology* 32 (2023), 1–61. doi:10.1145/3542945

[6] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. 2017. Formal Design and Verification of Self-Adaptive Systems with Decentralized Control. *ACM Transactions on Autonomous and Adaptive Systems* 11 (2017), 1–35. doi:10.1145/3019598

[7] Wesley K. G. Assunção, Luciano Marchezan, Lawrence Arkoh, Alexander Egyed, and Rudolf Ramler. 2024. Contemporary Software Modernization: Strategies, Driving Forces, and Research Opportunities. *ACM Transactions on Software Engineering and Methodology* 35 (2024), 1–35. doi:10.1145/3708527

[8] Ahmad Banijamali, Pasi Kuvaja, Markku Oivo, and Pooyan Jamshidi. 2020. Kuksa*: Self-adaptive Microservices in Automotive Systems. In *International Conference on Product-Focused Software Process Improvement (PROFES)*. Springer, 367–384. doi:10.1007/978-3-030-64148-1_23

[9] Luciano Baresi, Sam Guinea, and Giordano Tamburrelli. 2008. Towards decentralized self-adaptive component-based systems. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 57–64. doi:10.1145/1370018.1370029

[10] Philip A. Bernstein and Eric Newcomer. 2009. *Principles of Transaction Processing* (2 ed.). Morgan Kaufmann. doi:10.5555/1208930

[11] Sree Ram Boyapati and Claudia Szabo. 2022. Self-adaptation in Microservice Architectures: A Case Study. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 42–51. doi:10.1109/ICECCS54210.2022.00014

[12] Buoyant. 2026. *What is a Service Mesh?* Buoyant. https://www.buoyant.io/what-is-a-service-mesh Accessed: 2026-01-06.

[13] Phil Calçado. 2017. Pattern: Service Mesh. https://philcalcado.com/2017/08/03/pattern_service_mesh.html. Accessed: 2026-01-06.

[14] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola. 2012. MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems. *IEEE Transactions on Software Engineering* 38 (2012), 1138–1159. doi:10.1109/TSE.2011.68

[15] 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*. Vol. 5525. Springer, 1–26. doi:10.1007/978-3-642-02161-9_1

[16] João Pablo S. da Silva, Miguel Ecar, Marcelo S. Pimenta, Gilleanes T. A. Guedes, Luiz Paulo Franz, and Luciano Marchezan. 2018. A systematic literature review of UML-based domain-specific modeling languages for self-adaptive systems. In *International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 87–93. doi:10.1145/3194133.3194136

[17] Messias Filho, Eliaquim Pimentel, Wellington Pereira, Paulo Henrique M. Maia, and Mariela I. Cortés. 2021. Self-Adaptive Microservice-based Systems: Landscape and Research Opportunities. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 167–178. doi:10.1109/SEAMS51251.2021.00030

[18] Martin Fowler. 2014. Circuit Breaker. https://martinfowler.com/bliki/CircuitBreaker.html. Accessed: 2026-01-06.

[19] 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 (2004), 46–54. doi:10.1109/MC.2004.175

[20] Guillaume Gauvrit, Erwan Daubert, and Francoise Andre. 2010. SAFDIS: A Framework to Bring Self-Adaptability to Service-Based Distributed Applications. In *EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 211–218. doi:10.1109/SEAA.2010.25

[21] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. Applying Machine Learning in Self-adaptive Systems: A Systematic Literature Review. *ACM Transactions on Autonomous and Adaptive Systems* 15 (2021), 9:1–9:37. doi:10.1145/3469440

[22] Eli Gjørven, Romain Rouvoy, and Frank Eliassen. 2008. Cross-layer Self-adaptation of Service-oriented Architectures. In *Workshop on Middleware for Service Oriented Computing (MW4SOC)*. ACM, 37–42. doi:10.1145/1462802.1462809

[23] Thomas Glazier and David Garlan. 2019. An Automated Approach to Management of a Collection of Autonomic Systems. In *International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. IEEE, 110–115. doi:10.1109/FAS-W.2019.00038

[24] Thomas J. Glazier, David Garlan, and Bradley Schmerl. 2020. Automated Management of Collections of Autonomic Systems. In *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 82–91. doi:10.1109/ACSOS49614.2020.00029

[25] Hadaytullah, Sriharsha Vathsavayi, Outi Räihä, Kai Koskimies, and Allan Gregersen. 2012. Applying genetic self-architecting for distributed systems. In *Proceedings of the 2012 Fourth World Congress on Nature and Biologically Inspired Computing (NaBIC)*. IEEE, 44–52. doi:10.1109/NaBIC.2012.6402238

[26] 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. *Journal of Systems and Software* 85 (2012), 2840–2859. doi:10.1016/j.jss.2012.07.052

[27] Nikolaus Huber, André Hoorn, Anne Koziolek, Fabian Brosig, and Samuel Kounev. 2014. Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. *Service Oriented Computing and Applications* 8 (2014), 73–89. doi:10.1007/s11761-013-0144-4

[28] Mahmoud Imdoukh, Imtiaz Ahmad, and Mohammad Alfailakawi. 2020. Machine Learning Based Auto-Scaling for Containerized Applications. *Neural Computing and Applications* 32 (2020), 9745–9760. doi:10.1007/s00521-019-04507-z

[29] Istio Authors. 2026. Architecture. https://istio.io/latest/docs/ops/deployment/architecture/. Accessed: 2026-01-06.

[30] Nicholas R. Jennings. 2000. On Agent-based Software Engineering. *Artificial Intelligence* 117 (2000), 277–296. doi:10.1016/S0004-3702(99)00107-1

[31] Nicholas R. Jennings. 2001. An Agent-based Approach for Building Complex Software Systems. *Commun. ACM* 44 (2001), 35–41. doi:10.1145/367211.367250

[32] Stefan Kehrer and Wolfgang Blochinger. 2018. Model-Based Generation of Self-adaptive Cloud Services. In *International Conference on Cloud Computing and Services Science (CLOSER) (Communications in Computer and Information Science)*. Springer, 40–63. doi:10.1007/978-3-030-29193-8_3

[33] J.O. Kephart and D.M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36 (2003), 41–50. doi:10.1109/MC.2003.1160055

[34] Anne-Marie Kermarrec and Maarten van Steen. 2007. Gossiping in distributed systems. *SIGOPS Operating Systems Review* 41 (2007), 2–7. doi:10.1145/1317379.1317381

[35] Christian Krupitzer, Felix Maximilian Roth, Martin Pfannemüller, and Christian Becker. 2016. Comparison of Approaches for Self-Improvement in Self-Adaptive Systems. In *International Conference on Autonomic Computing (ICAC)*. IEEE, 308–314. doi:10.1109/ICAC.2016.18

[36] 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.

[37] Peini Liu, Xinjun Mao, Shuai Zhang, and Fu Hou. 2018. Towards Reference Architecture for a Multi-layer Controlled Self-adaptive Microservice System. In *International Conference on Software Engineering and Knowledge Engineering (SEKE)*. KSI Research Inc., 236–241.

[38] Markus Luckey and Gregor Engels. 2013. High-quality specification of self-adaptive software systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 143–152. doi:10.1109/SEAMS.2013.6595501

[39] Basel Magableh and Muder Almiani. 2019. A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster. In *Advanced Information Networking and Applications (AINA)*, Vol. 926. Springer, 846–858. doi:10.1007/978-3-030-15032-7_71

[40] Lukas Malburg, Maximilian Hoffmann, and Ralph Bergmann. 2023. Applying MAPE-K Control Loops for Adaptive Workflow Management in Smart Factories. *Journal of Intelligent Information Systems* 61 (2023), 83–111. doi:10.1007/s10844-022-00766-w

[41] Daniel A. Menascé, Hassan Gomaa, Sam Malek, and João Pedro Sousa. 2011. SASSY: A Framework for Self-Architecting Service-Oriented Systems. *IEEE Software* 28 (2011), 78–85. doi:10.1109/MS.2011.22

[42] Nabor C. Mendonca, Pooyan Jamshidi, David Garlan, and Claus Pahl. 2021. Developing Self-Adaptive Microservice Systems: Challenges and Directions. *IEEE Software* 38 (2021), 70–79. doi:10.1109/MS.2019.2955937

[43] Nabor C. Mendonça, David Garlan, Bradley Schmerl, and Javier Cámara. 2018. Generality vs. reusability in architecture-based self-adaptation: the case for self-adaptive microservices. In *European Conference on Software Architecture: Companion Proceedings (ECSA)*. ACM, 1–6. doi:10.1145/3241403.3241423

[44] 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 (2024), 1251–1272. doi:10.1007/s00607-022-01052-x

[45] Afaf Mousa, Jamal Bentahar, and Omar Alam. 2018. Multi-Objective Self-Adaptive Composite SaaS Using Feature Model. In *International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 77–84. doi:10.1109/FiCloud.2018.00019

[46] Nathalia Nascimento, Paulo Alencar, and Donald Cowan. 2023. Self-Adaptive Large Language Model (LLM)-Based Multiagent Systems. In *International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 104–109. doi:10.1109/ACSOS-C58168.2023.00048

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

[48] A. Nicolas-Plata, J. L. Gonzalez-Compean, and V. J. Sosa-Sosa. 2024. A service mesh approach to integrate processing patterns into microservices applications. *Cluster Computing* 27 (2024), 7417–7438. doi:10.1007/s10586-024-04342-5

[49] João Paulo Karol Santos Nunes, Shiva Nejati, Mehrdad Sabetzadeh, and Elisa Yumi Nakagawa. 2024. Self-Adaptive, Requirements-Driven Autoscaling of Microservices. In *Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 168–174. doi:10.1145/3643915.3644094

[50] Jiyoung Oh, Claudia Raibulet, and Joran Leest. 2023. Analysis of MAPE-K Loop in Self-adaptive Systems for Cloud, IoT and CPS. In *International Conference on Service-Oriented Computing (ICSOC)*. Springer, 130–141. doi:10.1007/978-3-031-26507-5_11

[51] Nuno Oliveira and Luís S. Barbosa. 2014. A Self-Adaptation Strategy for Service-Based Architectures. In *Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. IEEE, 1–10. doi:10.1109/SBCARS.2014.12

[52] Diego Perez-Palacin and José Merseguer. 2011. Performance sensitive self-adaptive service-oriented software using hidden Markov models. In *International Conference on Performance Engineering (ICPE)*, Vol. 36. ACM, 40–40. doi:10.1145/1958746.1958776

[53] Martin Pfannmüller, Martin Breitbach, Christian Krupitzer, Markus Weckesser, Christian Becker, Bradley Schmerl, and Andy Schürr. 2020. REACT: A Model-Based Runtime Environment for Adapting Communication Systems. In *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 65–74. doi:10.1109/ACSOS49614.2020.00027

[54] Harald Psaier, Lukasz Juszczyk, Florian Skopik, Daniel Schall, and Schahram Dustdar. 2010. Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems. In *International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 164–173. doi:10.1109/SASO.2010.44

[55] Ali Rahmanian, Ahmed Ali-Eldin, Björn Skubic, and Erik Elmroth. 2022. MicroSplit: Efficient Splitting of Microservices on Edge Clouds. In *Symposium on Edge Computing (SEC)*. IEEE, 252–264. doi:10.1109/SEC54971.2022.00027

[56] Etienne Rivière and Spyros Voulgaris. 2011. Gossip-Based Networking for Internet-Scale Distributed Systems. In *E-Technologies: Transformation in a Connected World*. Springer, 253–284. doi:10.1007/978-3-642-20862-1_18

[57] Adalberto R. Sampaio, Julia Rubin, Ivan Beschastnikh, and Nelson S. Rosa. 2019. Improving Microservice-based Applications with Runtime Placement Adaptation. *Journal of Internet Services and Applications* 10 (2019), 4:1–4:30. doi:10.1186/s13174-019-0104-0

[58] Brell Peclard Sanwouo Chekam, Clément Quinton, and Paul Temple. 2025. *Breaking the Loop: AWARE is the New MAPE-K.* ACM, 626–630. doi:10.1145/3696630.372851

[59] Komal Sarda, Zakeya Namrud, Marin Litoiu, Larisa Shwartz, and Ian Watts. 2024. Leveraging Large Language Models for the Auto-remediation of Microservice Applications: An Experimental Study. In *International Conference on the Foundations of Software Engineering (FSE).* ACM, 358–369. doi:10.1145/3663529.3663855

[60] Johanneke Siljee, Ivor Bosloper, Jos Nijhuis, and Dieter Hammer. 2005. DySOA: making service systems self-adaptive. In *International Conference on Service-Oriented Computing (ICSOC).* Springer-Verlag, 255–268. doi:10.1007/11596141_20

[61] Hongbign Wang, Xin Chen, Qin Wu, Qi Yu, Xingguo Hu, Zibin Zheng, and Athman Bouguettaya. 2017. Integrating Reinforcement Learning with Multi-Agent Techniques for Adaptive Service Composition. *ACM Transactions on Autonomous and Adaptive Systems* (2017), 1–42. doi:10.1145/3058592

[62] Sheng Wang, Zhijun Ding, and Changjun Jiang. 2021. Elastic Scheduling for Microservice Applications in Clouds. *IEEE Transactions on Parallel and Distributed Systems* 32 (2021), 98–115. doi:10.1109/TPDS.2020.3011979

[63] Teng Wang, Xiang He, Hanchuan Xu, Zhiying Tu, and Zhongjie Wang. 2021. EPF4M: An Evolution-Oriented Programming Framework for Microservices. In *International Conference on Services Computing (SCC).* IEEE, 174–182. doi:10.1109/SCC53864.2021.00030

[64] Danny Weyns, M. Usman Iftikhar, Danny Hughes, and Nelson Matthys. 2018. Applying Architecture-Based Adaptation to Automate the Management of Internet-of-Things. In *European Conference on Software Architecture (ECSA)*, Vol. 11048. Springer, 49–67. doi:10.1007/978-3-030-00761-4_4

[65] Danny Weyns and Usman M. Iftikhar. 2023. ActivFORMS: A Formally Founded Model-based Approach to Engineer Self-adaptive Systems. *ACM Transactions on Software Engineering and Methodology* 32 (2023), 1–48. doi:10.1145/3522585

[66] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. 2013. *On Patterns for Decentralized Control in Self-Adaptive Systems.* Springer, 76–107. doi:10.1007/978-3-642-35813-5_4

[67] Terence Wong, Markus Wagner, and Christoph Treude. 2022. Self-adaptive Systems: A Systematic Literature Review Across Categories and Domains. *Information and Software Technology* 148 (2022), 106934. doi:10.1016/j.infsof.2022.106934

[68] Shuai Zhang, Mingjiang Zhang, Lin Ni, and Peini Liu. 2019. A Multi-level Self-adaptation Approach for Microservice Systems. In *International Conference on Cloud Computing and Big Data Analysis (ICCCBDA).* IEEE, 498–502. doi:10.1109/ICCCBDA.2019.8725647