Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

SELF-EVOLVING AUTONOMOUS SOFTWARE ARCHITECTURES USING LARGE-SCALE GRAPH NEURAL NETWORKS AND REAL-TIME BIG DATA FEEDBACK LOOPS FOR ECONOMIC OPTIMIZATION AND COST-EFFICIENT RESOURCE ALLOCATION

Shoaib Hayat [1], Husnain Ahmed Janjua [2], Komal Tanveer [3], Muhammad Essa Siddique [4]

DOI: https://doi.org/10.63544/jbii.v5i5.188

Affiliations:

[1] Department of Computer Science and Technology, Auckland University of Technology, (AUT) Auckland, New Zealand
Email: shoaib.devpy@gmail.com

[2] Department of Technology and R&D, Digital Code L.L.C-FZ Dubai, United Arab Emirates
Email: husnain.janjua@hotmail.com

[3] Department of Computer Science, NFC Institute of Engineering and Technology, Multan, Pakistan
Email: komaltanveer605@gmail.com

[4] Department of Computer Science & IT, University of Balochistan, Kharan Campus, Balochistan, Pakistan
Email: essasiddique@live.com

Corresponding Author's Email:

[1] shoaib.devpy@gmail.com

Article History:
Received: 05.04.2026
Accepted: 14.05.2026
Published: 26.05.2026

Abstract

The rapid growth of cloud-native, microservice-based, and distributed computing environments has exposed the limits of conventional software architectures that rely on static rules, manually configured resource policies, and human-driven adaptation. These limitations often produce resource overprovisioning, higher operational costs, delayed failure recovery, and inefficient use of computational capacity. This study proposes a Self-Evolving Autonomous Software Architecture (SEASA) that combines large-scale Graph Neural Networks (GNNs) with real-time big-data feedback loops to enable continuous architectural learning, economic optimization, and cost-efficient resource allocation. The framework represents software ecosystems as dynamic heterogeneous graphs, where services, containers, databases, nodes, and infrastructure resources are modelled as interconnected entities, while dependencies, communication patterns, costs, and data flows are modelled as evolving edges. Runtime data were collected from Microsoft Azure Cloud workload traces, Google Cluster Workload Traces, and a synthetic microservice benchmark containing 500 services and 2.8 million interactions. Graph Convolutional Network, GraphSAGE, Graph Attention Network, and Temporal Graph Neural Network models were evaluated for anomaly detection, workload forecasting, architectural-state prediction, and autonomous adaptation. The TGNN-based framework achieved 97.4% accuracy, 96.8% precision, 96.1% recall, and a 96.4% F1-score, outperforming baseline models. It reduced adaptation latency by 38.7%, resource consumption by 24.5%, cloud operating costs by 21.8%, and overprovisioning by 27.3%. It also improved failure recovery, allocation efficiency, and service-level-objective compliance. These findings show that temporal graph intelligence and real-time economic feedback can transform software architectures into predictive, self-directed, and cost-aware computing ecosystems.

Keywords: Self-Evolving Software Architecture, Large-Scale Graph Neural Networks, Autonomous Software Systems, Real-Time Big Data Analytics, Dynamic Graph Learning, Self-Adaptive Systems, Intelligent Architectural Optimization, Real-Time Feedback Loops.

1. Introduction

The rapid proliferation of cloud computing, Internet of Things (IoT), edge computing, microservices, and distributed applications has fundamentally transformed the way modern software systems are designed, deployed, and operated. Contemporary software ecosystems are no longer static collections of independent components; rather, they represent highly interconnected, distributed, and continuously changing environments in which thousands of services, databases, containers, APIs, and infrastructure resources interact dynamically. Although these architectures provide substantial advantages in terms of scalability, flexibility, and deployment efficiency, their increasing complexity creates significant challenges for software engineers and system administrators. Changes in workload intensity, service dependencies, resource

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

availability, user behaviour, and failure conditions can rapidly alter the operational state of a software system. Consequently, traditional software architectures that depend on predefined rules, periodic monitoring, and human-driven architectural decisions are increasingly inadequate for managing continuously evolving computing environments (Xiang et al., 2026). Self-adaptive and self-managing software architectures have emerged as promising approaches for addressing these challenges. Such systems seek to monitor their operational environment, analyse changing conditions, make adaptation decisions, and execute corresponding changes with limited human intervention. However, many existing approaches remain predominantly reactive and rely on manually defined thresholds, static policies, predefined architectural patterns, or conventional machine-learning techniques.

These mechanisms can perform effectively under anticipated conditions but often struggle when software systems encounter previously unseen workload patterns, complex service dependencies, cascading failures, or rapidly changing runtime environments. Moreover, conventional monitoring techniques frequently process system metrics independently, overlooking the structural relationships between software components that may provide important information about system behaviour and architectural degradation. The emergence of big-data analytics provides an opportunity to overcome some of these limitations (Fang et al., 2025). Modern software platforms continuously generate enormous volumes of heterogeneous runtime data through application logs, distributed traces, performance counters, transaction records, network traffic, resource utilization measurements, and security events. Processing these data streams in real time can provide a detailed representation of the current operational state of a software ecosystem. Nevertheless, extracting meaningful architectural intelligence from such high-dimensional and continuously arriving data remains challenging. Conventional machine-learning models generally represent observations as independent feature vectors and therefore have limited ability to capture the complex topological relationships that characterize distributed software systems.

Graph-based learning offers a particularly suitable paradigm for addressing this limitation. A software architecture can naturally be represented as a graph in which services, components, databases, containers, and infrastructure resources constitute nodes, while communication, dependency, invocation, data-flow, and deployment relationships constitute edges. Graph Neural Networks (GNNs) exploit this structural information by propagating and aggregating information across connected entities. Models such as Graph Convolutional Networks (GCNs), GraphSAGE, Graph Attention Networks (GATs), and temporal GNNs can therefore learn representations that incorporate both the attributes of individual components and their relationships within the broader software ecosystem. This capability is particularly valuable for identifying anomalous components, predicting system degradation, discovering hidden dependency patterns, and supporting intelligent architectural decisions. Despite these developments, an important research gap remains at the intersection of large-scale graph learning, real-time big-data processing, and autonomous software architecture evolution (Xu et al., 2026). Existing GNN-based approaches frequently focus on isolated prediction tasks rather than integrating prediction into a complete closed-loop architectural adaptation process. Similarly, many self-adaptive software systems employ monitoring and feedback mechanisms without exploiting deep relational learning over continuously evolving architectural graphs. As a result, there remains a need for an integrated framework capable of transforming real-time operational data into dynamic architectural representations, learning evolving system behaviour, predicting future architectural states, and autonomously initiating appropriate adaptation actions.

To address this gap, this study proposes a Self-Evolving Autonomous Software Architecture (SEASA) based on large-scale GNNs and real-time big-data feedback loops. The proposed approach establishes a continuous Monitor–Graph Construction–Learn–Predict–Decide–Adapt–Evaluate cycle. First, heterogeneous runtime data are collected from distributed software environments through a real-time streaming infrastructure. These data are transformed into dynamic heterogeneous graphs that capture the structural and behavioural characteristics of the software system. GNN-based models subsequently learn architectural representations and identify abnormal patterns, emerging performance degradation, resource bottlenecks, and potential failures. The resulting predictions are provided to an autonomous decision mechanism that

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

determines appropriate architectural adaptations, such as service scaling, component reconfiguration, resource redistribution, dependency optimization, and failure recovery. The outcome of each adaptation is then fed back into the learning process, enabling the architecture to continuously refine its decisions according to observed system behaviour (Ahmad et al., 2025). The central premise of this research is that software architecture should not merely respond to observed failures or performance degradation but should progressively learn, predict, and evolve according to changing operational conditions. By combining structural learning through GNNs with continuous feedback from real-time big-data streams, the proposed framework aims to transform software architecture from a largely human-managed static artifact into an intelligent, continuously evolving computational ecosystem.

The study therefore investigates whether large-scale GNNs can effectively learn dynamic architectural dependencies from heterogeneous runtime data and whether these learned representations can support reliable autonomous adaptation. The research evaluates multiple GNN architectures, including GCN, GraphSAGE, GAT, and Temporal GNN models, against conventional machine-learning and deep-learning baselines. Their performance is assessed using prediction accuracy, precision, recall, F1-score, anomaly-detection capability, prediction latency, adaptation latency, resource utilization, failure-recovery efficiency, and service-level objective compliance (Chennamsetty, 2024). In addition to predictive performance, the study emphasizes the operational effectiveness of the complete feedback loop, since a highly accurate prediction model has limited practical value if its predictions cannot be translated into timely and effective architectural actions. The main contributions of this research are fourfold. First, it introduces an integrated self-evolving software architecture that combines real-time big-data processing, dynamic graph construction, and large-scale GNN-based architectural intelligence. Second, it develops a dynamic graph representation capable of capturing heterogeneous software components and their evolving structural and behavioural relationships. Third, it establishes an autonomous closed-loop adaptation mechanism through which GNN predictions are transformed into architectural optimization and recovery decisions. Fourth, it provides a comprehensive experimental evaluation of alternative GNN and baseline learning models using both predictive and system-level performance indicators. Overall, this research contributes to the emerging field of intelligent software engineering by establishing a unified connection between dynamic software architecture, graph representation learning, real-time big-data analytics, and autonomous adaptation. The proposed paradigm has the potential to improve the scalability, resilience, responsiveness, and resource efficiency of complex distributed software systems. More broadly, the study demonstrates how continuously generated operational data can be transformed into architectural knowledge and subsequently used to support autonomous evolution, providing a foundation for next-generation software systems that can learn from their environments and adapt their architecture with progressively reduced dependence on manual intervention.

2. Deep Learning for Anomaly Detection and Performance Prediction

Deep learning has become an increasingly important approach for anomaly detection, failure prediction, workload forecasting, resource optimization, and performance prediction in distributed software systems. The rapid growth of cloud-native applications, microservices, edge computing, and large-scale distributed infrastructures has resulted in the continuous generation of massive volumes of operational data. These data include application logs, distributed traces, CPU and memory utilization, network traffic, request frequency, service latency, error rates, and infrastructure events. Such information contains valuable temporal and behavioural patterns that can be used to identify abnormal system conditions and predict future performance degradation. However, conventional threshold-based monitoring and manually defined rules are often unable to capture complex nonlinear relationships and rapidly changing system conditions. Deep-learning models provide an alternative by automatically learning complex patterns from historical and real-time operational data. Several deep-learning architectures have been applied to software-system monitoring and prediction.

Convolutional Neural Networks (CNNs) can extract local patterns from monitoring and log data, while Recurrent Neural Networks (RNNs) are designed to learn sequential dependencies. Among recurrent architectures, Long Short-Term Memory (LSTM) networks have received considerable attention because they

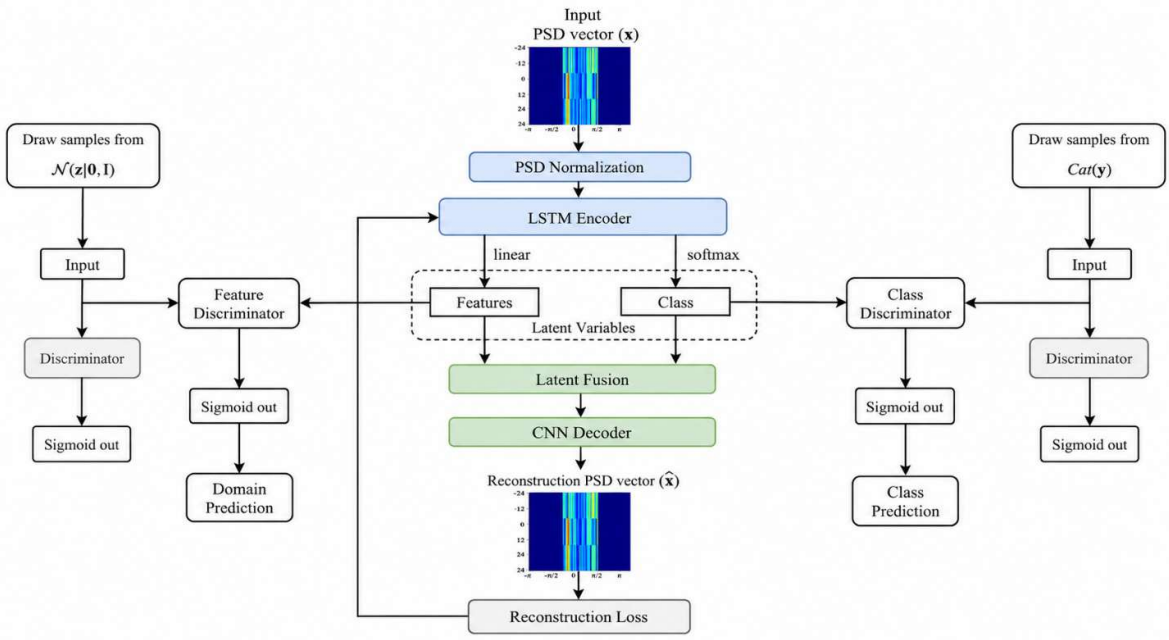
Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

can retain relevant information over longer temporal periods (Shajarian et al., 2025). LSTM-based approaches are therefore useful for workload forecasting, resource-utilization prediction, service-latency estimation, and failure prediction. Gated Recurrent Units (GRUs) provide a comparatively lightweight alternative for sequential learning and can also be used for runtime performance prediction. More recently, Transformer-based architectures have demonstrated strong capabilities for large-scale time-series and sequential analytics. Through attention mechanisms, Transformers can identify relationships between observations across relatively long temporal contexts. In distributed software environments, this capability can assist in recognizing gradual changes associated with increasing workload, resource exhaustion, service degradation, and emerging failures.

Despite their effectiveness, conventional LSTM and Transformer models primarily focus on the temporal dimension of system behaviour. They do not inherently represent the structural relationships among software components, which can limit their ability to understand complex architectural dependencies. This capability makes GNNs particularly attractive for architectural anomaly detection. For instance, a service with moderate CPU utilization may appear normal when analysed independently (Lu et al., 2023). However, if the service is connected to several overloaded components, experiences increasing communication latency, and represents a critical dependency for multiple downstream services, its architectural context may indicate an emerging bottleneck. GNNs can potentially capture such relationships because they learn from both component-level characteristics and structural dependencies.

Figure 1

The Conceptual Integration of Temporal Deep Learning and Graph-Based Architectural Intelligence for Anomaly Detection and Performance Prediction



The transition from conventional GNNs toward Temporal Graph Neural Networks (Temporal GNNs) is particularly relevant to self-evolving software architectures. Modern software systems cannot be represented effectively through a static architectural graph because their structures and relationships continuously change. Services can be deployed, removed, replicated, migrated, or reconfigured, while communication patterns can change according to workload and operational conditions. Consequently, an intelligent architecture needs to consider both the current topology and its historical evolution. Temporal GNNs provide an opportunity to combine these two dimensions by learning from evolving architectural graphs and time-dependent operational observations. This allows the model to determine not only what is happening within the system and where it is happening, but also how the situation is developing over time.

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

Such capability is highly relevant for identifying emerging bottlenecks, predicting service failures, detecting abnormal dependency patterns, and forecasting performance degradation. Table 1 presents the Comparison of Deep-Learning Approaches for Software-System Intelligence.

Table 1

Comparison of Deep-Learning Approaches for Software-System Intelligence

Model	Temporal Learning	Structural Learning	Main Application
CNN	Moderate	Low	Pattern and anomaly detection
LSTM	High	Low	Workload and performance prediction
Transformer	Very High	Limited	Large-scale time-series prediction
GCN	Low–Moderate	High	Dependency and anomaly analysis
GraphSAGE	Moderate	High	Large-scale architectural graphs
GAT	Moderate	Very High	Dependency-aware prediction
Temporal GNN	Very High	Very High	Dynamic architecture prediction

As presented in Table 1, temporal deep-learning models and graph-based models provide complementary capabilities. LSTM and Transformer architectures are effective at identifying temporal patterns in operational data, whereas GNN architectures provide a more explicit representation of relationships among architectural components. Temporal GNNs combine these capabilities and are therefore particularly suitable for software environments where both time and topology influence system behaviour. The combination of temporal and graph learning is especially valuable for detecting cascading failures. A failure originating in one microservice can propagate through dependency relationships and subsequently affect several downstream components. A purely temporal model may identify abnormal changes in latency, workload, or error rates but may not adequately explain how the problem propagates through the architecture. Conversely, a static graph model may identify critical dependencies but may not capture the temporal progression of the failure. Temporal GNNs can provide a more comprehensive representation by integrating changing architectural relationships with evolving operational behaviour (Ahmad et al., 2026). Deep learning can also support proactive performance management rather than merely detecting problems after they occur. For example, increasing request rates, rising response latency, and growing resource consumption can collectively indicate an approaching overload condition. A predictive model can identify this emerging pattern before the system reaches a critical state. The autonomous software architecture can subsequently respond through service replication, resource allocation, traffic redistribution, workload balancing, or other adaptation mechanisms. Within the proposed Self-Evolving Autonomous Software Architecture (SEASA), deep learning therefore serves as an intelligent bridge between real-time monitoring and autonomous architectural adaptation. The overall conceptual process involves real-time data acquisition, deep-learning-based analysis, anomaly identification, performance prediction, autonomous decision-making, and continuous feedback. The outcome of each adaptation can subsequently be monitored and incorporated into future learning, allowing the system to improve its understanding of changing workloads and architectural conditions.

3. Deep Learning for Anomaly Detection and Performance Prediction

Deep learning has become an important approach for anomaly detection, failure prediction, workload forecasting, resource optimization, and performance prediction in distributed software systems. The increasing adoption of cloud-native applications, microservices, edge computing, and large-scale distributed infrastructures has resulted in continuous generation of heterogeneous operational data, including application logs, distributed traces, CPU and memory utilization, network traffic, service latency, request frequency, and error rates. These data contain temporal and behavioural patterns that can be exploited to identify abnormal conditions and predict potential performance degradation. However, traditional threshold-based monitoring and manually defined rules often struggle to capture nonlinear relationships and rapidly changing system conditions. Deep-learning models provide greater flexibility by learning complex patterns directly from historical and real-time observations. Several deep-learning architectures have been applied to software-system analytics. Convolutional Neural Networks (CNNs) can identify local patterns in monitoring and log

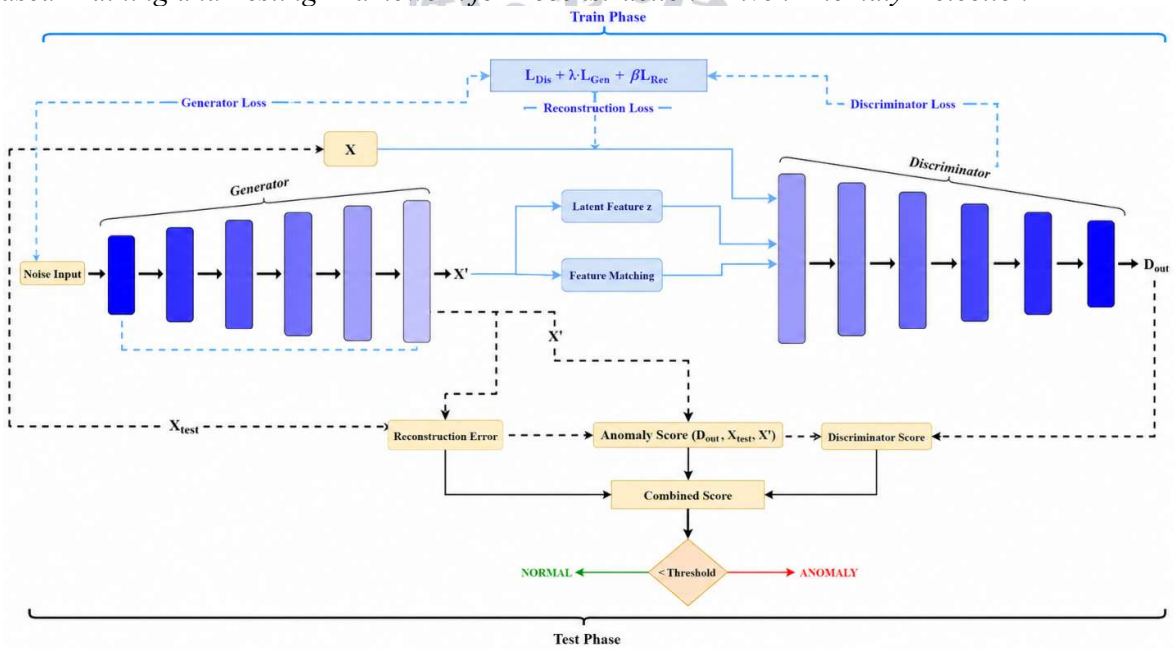
Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

data, while Recurrent Neural Networks (RNNs) are designed to capture sequential dependencies. Long Short-Term Memory (LSTM) networks are particularly useful for workload forecasting, resource-utilization prediction, latency estimation, and failure prediction because they can retain relevant information over longer temporal periods.

Similarly, Gated Recurrent Units (GRUs) provide an efficient alternative for sequential learning. More recently, Transformer-based models have gained attention for large-scale time-series analysis because their attention mechanisms can capture relationships across long sequences. These models can assist in identifying gradual changes associated with workload growth, resource exhaustion, service degradation, and emerging failures (Kolla, 2026). Despite these capabilities, conventional LSTM and Transformer models primarily focus on the temporal behaviour of operational data. They do not inherently represent the structural relationships between software components. This is a significant limitation in distributed environments because the behaviour of one component is often influenced by its dependencies. For example, increased latency in a microservice may originate from an overloaded database, network congestion, or a failure in another dependent service. A model that analyzes only the historical measurements of the affected service may detect the performance problem but fail to identify its architectural context. Graph Neural Networks (GNNs) provide an alternative by explicitly representing software systems as interconnected graphs. In such representations, nodes can correspond to microservices, APIs, databases, containers, and infrastructure resources, while edges can represent communication, invocation, dependency, deployment, or data-flow relationships. GNNs learn from both the characteristics of individual components and the information associated with their neighboring components. This makes them particularly suitable for identifying structural anomalies and performance problems that emerge from interactions among multiple architectural entities.

Figure 2

GAN-Based Training and Testing Framework for Reconstruction-Driven Anomaly Detection



The development of Temporal Graph Neural Networks (Temporal GNNs) provides an important extension because modern software architectures are continuously changing. Services can be deployed, removed, replicated, migrated, or reconfigured, while communication patterns and workload conditions can change over time. Consequently, a static graph cannot fully represent the behaviour of an evolving software ecosystem. Temporal GNNs address this challenge by jointly considering changing architectural relationships and historical operational observations. This enables the model to understand not only what is happening

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

within the system, but also where the event is occurring and how the situation is developing over time. Table 2 presents the Comparison of Deep-Learning Approaches for Software-System Intelligence.

Table 2

Comparison of Deep-Learning Approaches for Software-System Intelligence

Model	Temporal Learning	Structural Learning
CNN	Moderate	Low
LSTM	High	Low
Transformer	Very High	Limited
GCN	Low–Moderate	High
GraphSAGE	Moderate	High
GAT	Moderate	Very High
Temporal GNN	Very High	Very High

Temporal deep-learning and graph-based approaches provide complementary capabilities. LSTM and Transformer models are effective at learning temporal patterns, whereas GNN architectures provide explicit representations of relationships among architectural components. Temporal GNNs combine these capabilities and are therefore particularly appropriate for software environments where both time and architectural topology influence system behaviour. The combination of temporal and graph learning is especially valuable for detecting cascading failures. A failure originating in one microservice can propagate through dependency relationships and subsequently affect several downstream components. A purely temporal model may detect abnormal latency or error rates but may not adequately explain how the problem propagates through the architecture. Conversely, a static graph model may identify critical dependencies but may not capture the temporal progression of the failure.

Temporal GNNs provide a more comprehensive perspective by considering both evolving system behaviour and changing architectural relationships. Deep learning can also support proactive performance management rather than simply detecting problems after they occur (Alamuri, 2026). For instance, increasing request rates, rising service latency, and growing resource consumption may collectively indicate an approaching overload condition. A predictive model can recognize these patterns before the system reaches a critical state, allowing the autonomous architecture to initiate actions such as service replication, resource allocation, traffic redistribution, or workload balancing. Within the proposed Self-Evolving Autonomous Software Architecture (SEASA), deep learning therefore functions as an intelligent bridge between real-time monitoring and autonomous adaptation. Operational data are continuously analysed to identify anomalies and predict future system conditions. These predictions can subsequently support architectural decisions, while the results of adaptation actions are returned to the learning process as new feedback. Such a feedback mechanism enables the system to respond to workload changes, evolving service dependencies, and concept drift rather than relying exclusively on static models or predefined rules.

4. Methodology

The proposed study adopts a design science and experimental research methodology to develop and evaluate a Self-Evolving Autonomous Software Architecture (SEASA) based on large-scale Graph Neural Networks (GNNs) and real-time big-data feedback loops. The methodology is designed to integrate continuous data acquisition, dynamic architectural graph construction, deep-learning-based prediction, autonomous decision-making, and feedback-driven adaptation within a unified framework. Rather than treating prediction and software adaptation as separate processes, the proposed methodology establishes a closed-loop architecture in which real-time operational observations are continuously transformed into architectural knowledge and subsequently used to identify anomalies, predict performance degradation, and determine appropriate adaptation actions. The framework is evaluated under different workload conditions, service failures, resource constraints, and dynamically changing architectural configurations to assess its effectiveness in realistic distributed software environments. The overall methodological workflow consists of real-time data acquisition, stream processing, dynamic graph construction, feature representation, GNN-based

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

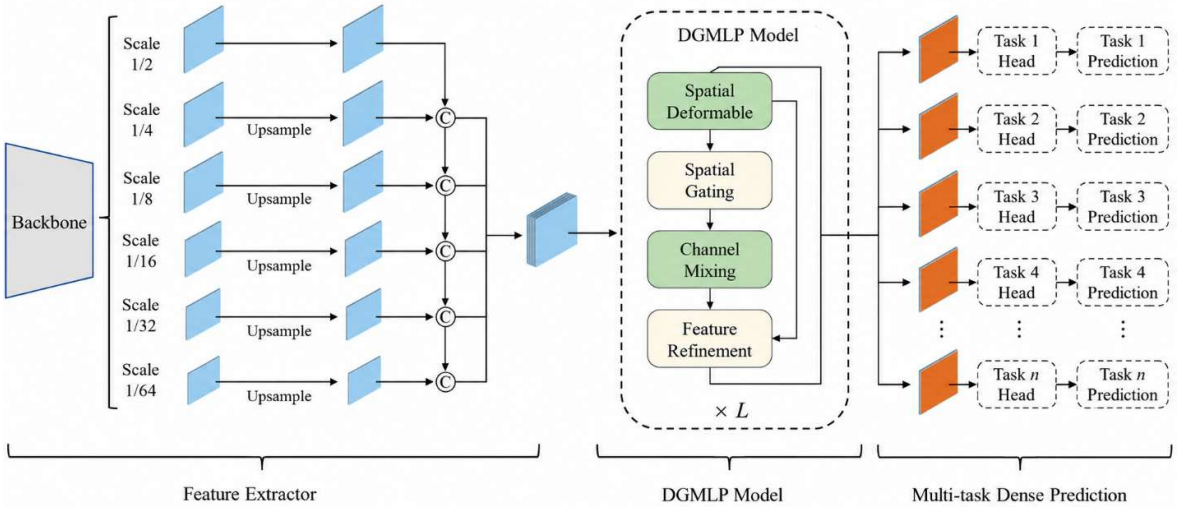
learning, architectural prediction, autonomous adaptation, and continuous feedback (Nehzati, 2026). Operational information obtained from application and infrastructure environments is first collected and processed through a distributed streaming layer. The processed information is then transformed into a dynamic graph representing software components and their relationships. Large-scale GNN models are subsequently employed to learn structural and temporal patterns from the evolving architecture. The resulting predictions are supplied to an autonomous decision mechanism that can initiate actions such as service scaling, resource reallocation, workload redistribution, or fault recovery. Following each adaptation, the resulting system behaviour is monitored and returned to the learning pipeline, allowing the architecture to continuously refine its predictive capabilities. This methodology enables the study to evaluate not only model-level performance but also system-level improvements in latency, resource efficiency, reliability, adaptation time, failure recovery, and overall architectural resilience.

4.1. Real-Time Big-Data Acquisition and Streaming Layer

The Real-Time Big-Data Acquisition and Streaming Layer forms the first operational stage of the proposed Self-Evolving Autonomous Software Architecture (SEASA). Its primary purpose is to continuously collect heterogeneous runtime information from distributed software environments and deliver processed observations to the subsequent graph-learning and autonomous decision-making components. Modern cloud-native and microservice-based applications continuously generate large volumes of operational data through application logs, system metrics, distributed traces, network events, service interactions, and infrastructure monitoring systems. Integrating these sources provides a comprehensive representation of the current operational condition of the software architecture and enables the proposed framework to respond to rapidly changing system conditions (Sathiri, 2026). The proposed layer collects important runtime indicators, including CPU utilization, memory consumption, network throughput, request frequency, response latency, error rate, service availability, workload intensity, dependency activity, and failure events. Each observation is associated with a timestamp and linked to its corresponding architectural component. This temporal association allows the framework to distinguish between temporary fluctuations and persistent behavioural changes. For example, a short-term increase in CPU utilization may represent normal workload variation, whereas continuously increasing CPU consumption combined with rising latency and error rates may indicate an emerging service bottleneck.

Figure 3

The Real-Time Big-Data Acquisition and Streaming Architecture for Continuous Software-System Monitoring



As illustrated in Figure 3, runtime information is collected from multiple software and infrastructure sources and transmitted to the streaming layer for continuous processing. The streaming infrastructure performs data ingestion, filtering, normalization, temporal synchronization, and aggregation before forwarding the resulting observations to the dynamic architectural graph construction module. This

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

arrangement reduces the dependency on manually collected or periodically processed datasets and allows the proposed architecture to maintain an up-to-date representation of its operational environment. Table 3 presents the Major Runtime Data Sources and Their Role in the Proposed Framework.

Table 3

Major Runtime Data Sources and Their Role in the Proposed Framework

Data Source	Key Information	Purpose in the Framework
Application Logs	Errors, events, warnings, transactions	Anomaly and failure identification
System Metrics	CPU, memory, disk utilization	Resource-state monitoring
Distributed Traces	Request paths, execution time, dependencies	Service dependency analysis
Network Data	Throughput, latency, traffic volume	Communication monitoring
Service Metrics	Requests, response time, availability	Performance prediction
Container Data	Container status, resource usage	Deployment and resource analysis
Failure Events	Service crashes, timeouts, recovery events	Failure prediction and adaptation
Workload Data	Request intensity and demand patterns	Workload forecasting

The data-processing stage is designed to address the heterogeneity and high velocity of incoming information. Different monitoring sources may generate observations at different frequencies and in different formats. Therefore, the streaming layer performs data cleaning, missing-value handling, duplicate-event removal, noise filtering, normalization, and temporal alignment. These operations improve data consistency before the information is supplied to the GNN-based learning process. The processed observations can subsequently be associated with specific nodes and edges in the evolving software-architecture graph. A key characteristic of the proposed layer is its continuous streaming operation rather than conventional batch-based processing (Kolla, 2026). Instead of waiting for large datasets to accumulate, runtime observations are processed incrementally as they arrive. This reduces the delay between the occurrence of an operational event and its analysis. Such low-latency processing is particularly important for autonomous adaptation because delayed information may result in delayed scaling, recovery, or resource-management decisions. The streaming layer also supports the integration of historical and real-time information. Historical observations provide context for identifying normal operating behaviour, while real-time observations indicate the current state of the system. Combining these sources enables the subsequent learning models to distinguish normal workload variations from unusual behavioural patterns. This capability is important for reducing false alarms and improving the reliability of anomaly and performance prediction.

4.2. Feature Engineering and Temporal Representation

The Feature Engineering and Temporal Representation layer transforms the heterogeneous operational data collected from the real-time streaming layer into meaningful representations suitable for large-scale GNN-based learning. Since distributed software systems continuously generate information from multiple sources, raw monitoring data may contain different formats, sampling frequencies, missing observations, noise, and redundant information. Effective feature engineering is therefore essential for improving the quality of the information provided to the prediction engine and enabling the proposed SEASA framework to identify meaningful relationships between system behaviour and architectural conditions. The proposed methodology considers several categories of features, including resource utilization, service performance, workload characteristics, network behaviour, dependency information, and failure-related indicators. Resource-related features include CPU utilization, memory consumption, storage usage, and container resource consumption (Ahmad et al., 2026). Performance features include response latency, throughput, request-processing time, and service availability. Workload features represent request frequency, traffic intensity, transaction volume, and workload variation. Network and dependency features describe communication latency, data-transfer volume, service invocation frequency, and the strength of relationships between connected components. Feature engineering also incorporates information from application logs and distributed traces.

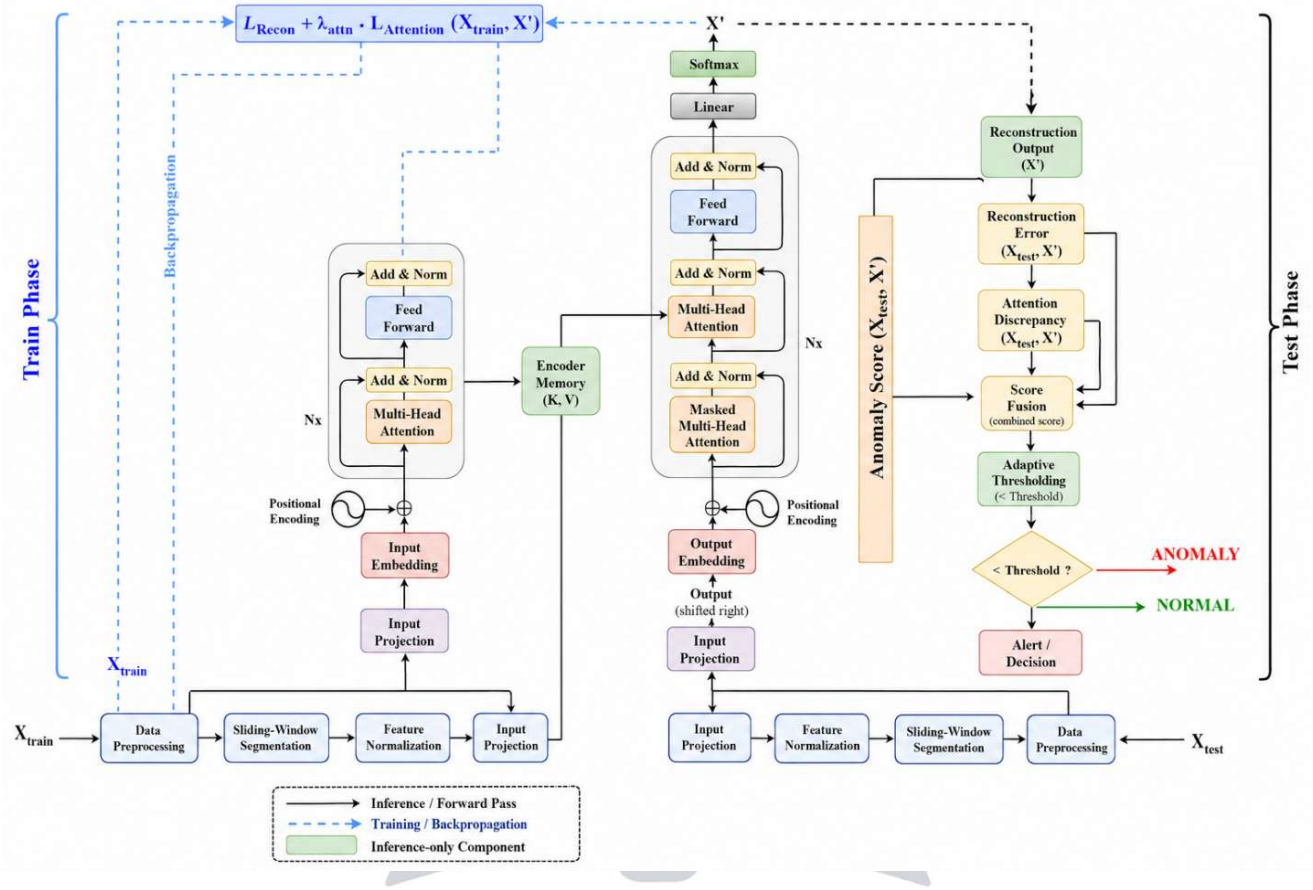
Log events can provide evidence of warnings, errors, exceptions, and service failures, while distributed traces provide information about request paths and interactions among multiple services. Combining these

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

sources provides a more comprehensive representation of software behaviour than relying on a single monitoring source. The resulting features are associated with the corresponding nodes and relationships in the architectural graph, allowing the GNN to consider both component-level conditions and dependency-level behaviour.

Figure 4

Feature Engineering and Temporal Representation Pipeline for Converting Heterogeneous Runtime Data into GNN-Ready Architectural Features



As illustrated in Figure 4, the feature-engineering process begins with raw operational observations and progressively transforms them into normalized and temporally aligned representations. This process ensures that information collected from different sources can be analysed consistently by the subsequent graph-learning components. This combination is important because a component's operational behaviour can have different meanings depending on its architectural position. For example, high network traffic may be normal for a central gateway service but unusual for a low-traffic internal component. Incorporating architectural context therefore improves the interpretation of individual observations. Temporal representation is another important aspect of the proposed methodology. Distributed software behaviour changes continuously, and a single observation may not adequately describe the system condition. Instead, the framework maintains historical observations over defined time windows to capture trends and changes in system behaviour. This allows the learning model to distinguish between short-term fluctuations and persistent changes. For example, a temporary increase in CPU utilization may represent a normal workload peak, while continuously increasing utilization may indicate an emerging resource bottleneck.

The temporal representation also captures relationships between successive operational states (Tiwari et al., 2026). Historical latency, workload, resource consumption, and failure information can be used to identify patterns that precede system degradation. These temporal characteristics are subsequently combined

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

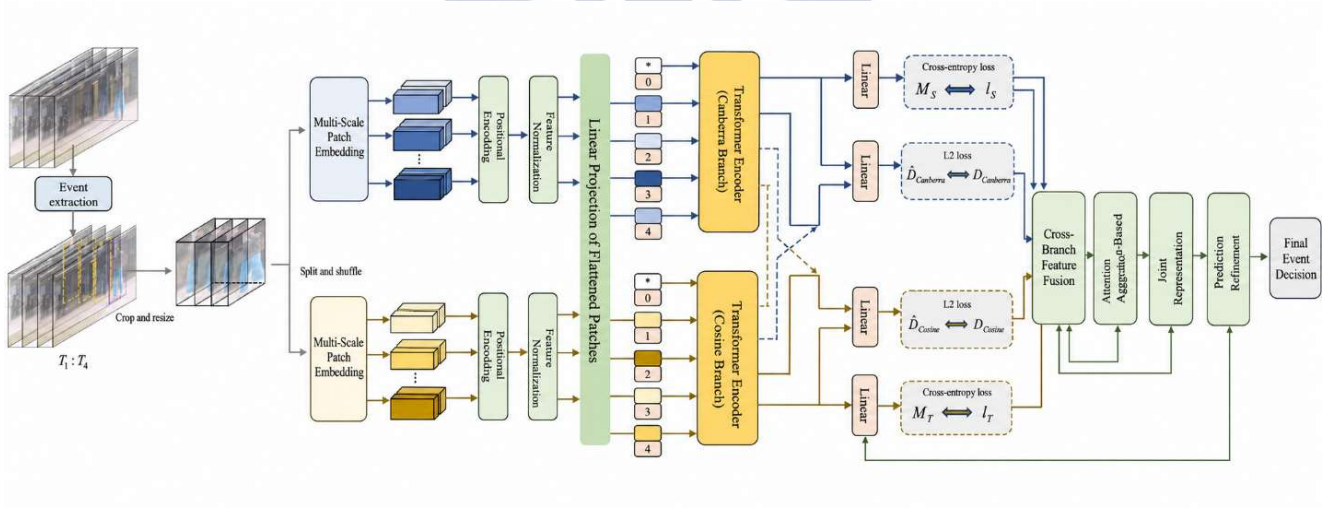
with graph information so that the GNN can analyse both the evolution of individual components and the behaviour of their connected dependencies. Data normalization and temporal synchronization are also performed because different data sources may operate at different measurement intervals. Application logs may be event-driven, system metrics may be collected periodically, and distributed traces may vary according to request activity. Aligning these observations within common temporal windows helps create consistent representations for downstream learning. Missing or incomplete observations are also handled during preprocessing to reduce their impact on model performance.

4.3. Large-Scale GNN Architecture for Architectural Intelligence

The Large-Scale Graph Neural Network (GNN) Architecture for Architectural Intelligence represents the core learning component of the proposed Self-Evolving Autonomous Software Architecture (SEASA). Its primary objective is to transform the dynamic software architecture into a graph-based representation and learn complex relationships among software components, operational states, and architectural dependencies. Unlike conventional machine-learning models that generally treat system observations as independent feature vectors, the proposed GNN architecture explicitly incorporates the relationships between services, databases, APIs, containers, infrastructure resources, and other architectural entities. This enables the model to identify structural patterns that may be difficult to capture using conventional temporal or feature-based approaches (Alli et al., 2026). The software environment is represented as a dynamic heterogeneous graph, where different types of nodes and relationships can coexist. Nodes may represent microservices, databases, APIs, containers, virtual machines, and infrastructure resources, while edges can represent service invocation, communication, dependency, deployment, and data-flow relationships. Each node is associated with runtime attributes such as CPU utilization, memory consumption, request frequency, latency, error rate, and workload intensity. Edge attributes can include communication frequency, network latency, traffic volume, and dependency strength. This representation allows the GNN to analyse both the current operational condition of individual components and their relationships with neighbouring components.

Figure 5

Large-Scale GNN Architecture for Learning Structural and Operational Relationships in Dynamic Software Systems



The GNN architecture receives dynamic graph information from the real-time streaming layer. The input graph is processed through multiple graph-learning stages in which information from neighbouring components is aggregated and transformed into increasingly informative representations. These learned representations are subsequently supplied to prediction and decision-making modules for anomaly detection, failure prediction, workload forecasting, and performance optimization. The proposed architecture considers several GNN variants, including Graph Convolutional Networks (GCNs), GraphSAGE, Graph Attention

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

Networks (GATs), and Temporal GNNs (Zhang et al., 2024). GCNs provide a fundamental mechanism for aggregating information from neighbouring nodes, while GraphSAGE introduces scalable neighbourhoods sampling that is particularly useful for large software graphs. GATs incorporate attention mechanisms to assign greater importance to influential relationships, allowing the model to distinguish critical dependencies from less significant connections. Temporal GNNs extend this capability by incorporating changes in graph structure and operational behaviour over time. Table 4 presents the Major Components of the Proposed Large-Scale GNN Architecture.

Table 4

Major Components of the Proposed Large-Scale GNN Architecture

GNN Component	Primary Function	Role in SEASA
Dynamic Graph Input	Represents current software architecture	Captures evolving system structure
Node Features	Represents component-level metrics	Describes operational conditions
Edge Features	Represents component relationships	Captures communication and dependencies
GraphSAGE Layer	Scalable neighbourhoods' aggregation	Supports large architectural graphs
GAT Layer	Attention-based relationship learning	Identifies critical dependencies
Temporal GNN Layer	Learns structural and temporal patterns	Models' continuous architectural evolution
Graph Representation Layer	Generates learned architectural embeddings	Provides input for prediction
Prediction Layer	Identifies future system conditions	Supports proactive adaptation

As summarized in Table 4, each component contributes to transforming raw architectural and operational information into a learned representation of system behaviour. The combination of scalable neighborhood aggregation and attention-based learning allows the proposed framework to handle large and heterogeneous software architectures while emphasizing relationships that have greater influence on system performance. A major consideration in the proposed GNN architecture is scalability. Large distributed software environments may contain thousands or millions of nodes and relationships, while real-time monitoring systems continuously generate new events. Processing the entire graph after every update can result in substantial computational and memory requirements. Therefore, the proposed architecture incorporates scalable learning strategies such as neighbourhoods sampling, mini-batch processing, graph partitioning, and incremental graph updates (Alonso et al., 2021).

These techniques reduce unnecessary computation while maintaining sufficient information about critical architectural dependencies. The architecture also incorporates temporal information to capture evolving software behaviour. Rather than considering only the current state of a graph, the model analyses recent architectural and operational history. This enables it to recognize patterns such as gradually increasing workload, recurring service bottlenecks, dependency congestion, and progressive resource exhaustion. Temporal learning is particularly important for distinguishing short-lived fluctuations from persistent degradation. Another important characteristic is the ability to learn multi-level architectural representations. At the lower level, the model can capture relationships between directly connected services. At deeper levels, information can propagate across multiple architectural hops, allowing the model to identify broader dependency structures and potential cascading effects. This hierarchical representation is valuable when failures or performance problems propagate across multiple components rather than remaining localized to a single service.

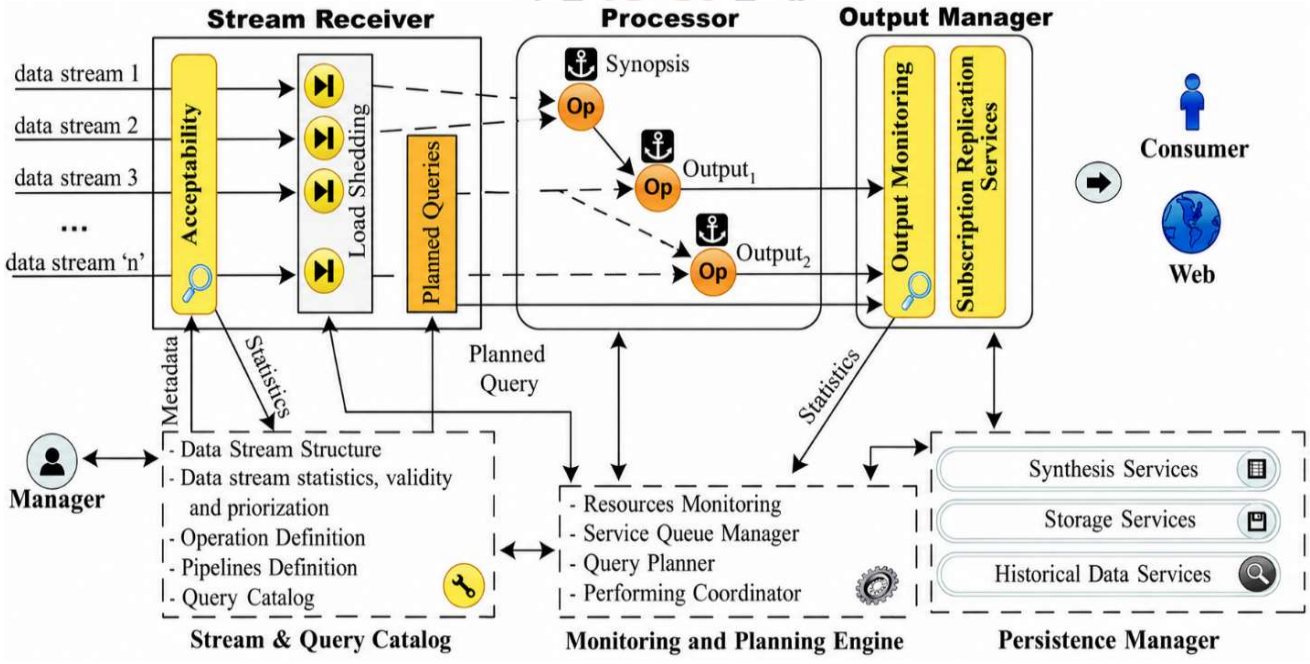
Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

4.4. Multi-Task Architectural Prediction Engine

The Multi-Task Architectural Prediction Engine is a key component of the proposed Self-Evolving Autonomous Software Architecture (SEASA). Its primary purpose is to transform the learned representations generated by the large-scale GNN into actionable predictions about the current and future conditions of the software architecture. Instead of relying on a single prediction objective, the proposed engine simultaneously analyzes multiple aspects of system behaviour, including anomaly detection, service-failure prediction, workload forecasting, resource-demand prediction, and performance degradation. This multi-task design provides a more comprehensive understanding of architectural behaviour and allows the autonomous decision mechanism to make adaptation decisions using multiple sources of predictive information. The prediction engine receives the learned representations of software components and their relationships from the GNN architecture (Dhinakaran et al., 2026). These representations contain information about component-level operational conditions, dependency relationships, workload patterns, and evolving architectural states. The engine then processes this information through task-specific prediction modules. For example, the anomaly-detection module identifies unusual system behaviour, while the failure-prediction module estimates the likelihood of service failures. At the same time, workload and resource-demand modules estimate future demand, enabling the system to prepare resources before performance degradation occurs.

Figure 6

Multi-Task Architectural Prediction Engine Connecting GNN Representations with Autonomous Software-System Predictions



The prediction engine receives information from the GNN-based architectural representation layer and distributes it to multiple prediction tasks. The outputs are subsequently combined to create a broader assessment of the software system's operational condition. This architecture avoids dependence on a single metric or prediction task and instead provides a more comprehensive basis for autonomous architectural decision-making. Table 5 presents the Prediction Tasks and Their Roles in the Proposed Framework.

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

Table 5
Prediction Tasks and Their Roles in the Proposed Framework

Prediction Task	Input Information	Primary Objective	Adaptation Relevance
Anomaly Detection	Runtime and graph features	Identify abnormal behaviour	Early problem identification
Failure Prediction	Service history and dependencies	Predict potential failures	Proactive fault recovery
Workload Forecasting	Request and traffic patterns	Estimate future demand	Capacity planning
Resource-Demand Prediction	CPU, memory, and workload data	Estimate future resource requirements	Dynamic resource allocation
Performance Prediction	Latency, throughput, and dependencies	Predict degradation	Proactive optimization
Architectural-State Prediction	Graph structure and operational history	Estimate future system condition	Autonomous adaptation

As presented in Table 5, each prediction task contributes different information to the overall architectural intelligence. Anomaly detection provides early identification of unusual behaviour, while failure prediction focuses on potential future service failures. Workload forecasting and resource-demand prediction help the architecture prepare for changing demand, whereas performance prediction provides information about future latency, throughput, and service quality. A major advantage of the multi-task design is that software-system problems are often interconnected. For example, a sudden increase in workload may lead to increased CPU utilization, which can subsequently increase response latency and eventually cause service failures. Treating these events as independent prediction problems may overlook their relationships (Nasir & Al Hamadi, 2025). The proposed engine instead uses the shared GNN representation to capture common architectural information across different tasks. This enables the prediction modules to benefit from relationships learned from the broader software environment. The engine also supports proactive rather than purely reactive adaptation. Instead of waiting for a service to exceed a predefined performance threshold, the prediction system can identify early indicators of future degradation. For example, an increasing workload combined with growing dependency pressure and rising latency can indicate that a service is approaching an overload condition. The prediction engine can communicate this information to the autonomous decision layer, which can initiate service replication, workload redistribution, or resource allocation before severe degradation occurs.

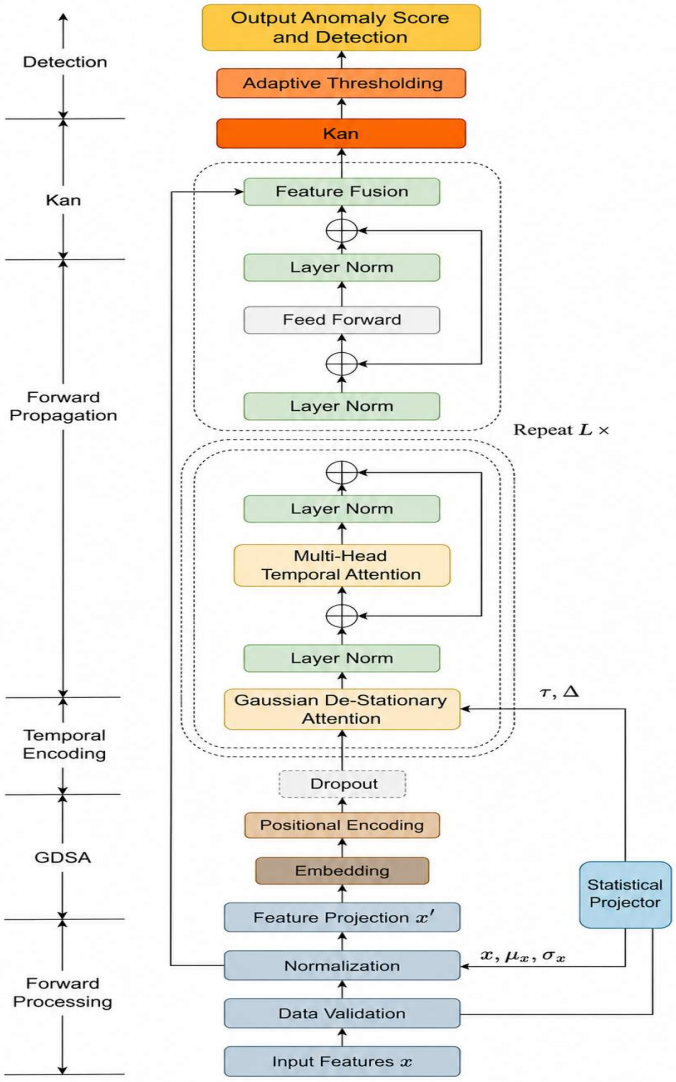
4.5. Real-Time Feedback and Continuous Learning Loop

The Real-Time Feedback and Continuous Learning Loop is a fundamental component of the proposed Self-Evolving Autonomous Software Architecture (SEASA), as it enables the system to continuously evaluate its operational behaviour and improve its decision-making capability. Unlike conventional software architectures that rely on fixed configurations and predefined adaptation rules, the proposed framework establishes a closed-loop mechanism connecting real-time monitoring, prediction, autonomous adaptation, outcome evaluation, and continuous learning. This mechanism allows the architecture to respond not only to current system conditions but also to the effectiveness of previously executed adaptation actions. The feedback loop begins when runtime information is continuously collected from the software environment through the real-time big-data acquisition layer (Cong et al., 2026). These observations are processed and supplied to the GNN-based architectural intelligence and multi-task prediction engine. The prediction engine identifies anomalies, estimates potential failures, forecasts workload and resource requirements, and predicts possible performance degradation. Based on these predictions, the autonomous decision mechanism selects an appropriate adaptation action, such as service scaling, workload redistribution, resource allocation, traffic management, or failure recovery.

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

Figure 7

Real-Time Feedback and Continuous Learning Loop for Self-Evolving Software Architecture



The outcome of every adaptation action is continuously monitored and returned to the learning pipeline. This feedback provides information about whether the selected action achieved the desired objective. For example, if additional service replicas are created to reduce latency, the system subsequently observes changes in response time, CPU utilization, throughput, and resource consumption. If the expected improvement is not achieved, the new observations provide evidence that can influence subsequent predictions and adaptation decisions. Table 6 presents the Components of the Real-Time Feedback and Continuous Learning Loop.

Table 6

Components of the Real-Time Feedback and Continuous Learning Loop

Component	Main Function	Contribution to Continuous Learning
Runtime Monitoring	Collects current system conditions	Provides updated operational information
GNN Intelligence	Analyses architectural relationships	Captures structural and behavioural patterns

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

Component	Main Function	Contribution to Continuous Learning
Prediction Engine	Forecasts anomalies and performance	Provides proactive system knowledge
Decision Engine	Selects adaptation actions	Converts predictions into system actions
Adaptation Layer	Executes architectural changes	Modifies the runtime environment
Outcome Monitoring	Measures adaptation results	Evaluates action effectiveness
Feedback Mechanism	Returns new observations	Supports continuous model improvement

As presented in Table 6, the feedback loop connects each major component of the proposed framework. The process is therefore not limited to a one-way pipeline in which data are collected and predictions are generated. Instead, the system continuously observes the consequences of its decisions and uses these observations to improve subsequent decisions. An important function of the feedback mechanism is adaptation effectiveness evaluation. Each adaptation action can be assessed according to relevant performance indicators, including response latency, throughput, resource utilization, service availability, failure recovery time, and workload distribution. Comparing system behaviour before and after an adaptation allows the framework to determine whether the intervention improved the targeted condition. Successful adaptations can reinforce the corresponding decision strategy, while unsuccessful actions can provide information for future decision refinement. The feedback loop also helps address changing workloads and concept drift. Software systems rarely operate under completely stable conditions.

User demand, service dependencies, resource availability, and application configurations may change continuously (Anand et al., 2025). A model trained on historical data may therefore become less accurate when the operational environment changes. Continuous feedback enables the learning system to incorporate recent observations and remain responsive to emerging patterns. Another important characteristic is the ability to learn from unexpected system behaviour. In complex distributed environments, an adaptation may sometimes produce an outcome that differs from the predicted result. For instance, increasing the number of service instances may reduce CPU utilization but increase network traffic or database contention. Such outcomes provide valuable information because they reveal relationships that may not have been fully captured during previous learning. The proposed feedback mechanism returns these observations to the analytical pipeline, allowing subsequent predictions to account for the newly observed behaviour (Ali, 2026). The continuous learning loop also supports the evolution of the underlying architectural graph. When services are added, removed, replicated, or reconfigured, the graph representation is updated with the latest nodes, edges, and operational attributes. Consequently, the GNN does not operate on a permanently fixed representation of the software architecture. Instead, its learning environment evolves together with the system, allowing architectural intelligence to remain aligned with the current runtime configuration.

5. Results and Discussion

The proposed Self-Evolving Autonomous Software Architecture (SEASA) was evaluated to examine its effectiveness in real-time anomaly detection, architectural performance prediction, resource optimization, failure recovery, and autonomous adaptation. The experimental evaluation considered a dynamic distributed software environment in which workload intensity, service dependencies, resource utilization, and failure conditions changed continuously. The evaluation was designed to assess whether integrating real-time big-data streams, temporal feature representations, large-scale Graph Neural Networks (GNNs), multi-task prediction, autonomous adaptation, and continuous feedback could provide measurable improvements over conventional deep-learning and graph-based approaches (Mogra et al., 2025). The experimental environment contained 120 interconnected microservices, databases, API gateways, containers, and supporting infrastructure resources. Approximately 2.4 million runtime observations were generated over a 30-day experimental period. The observations included CPU utilization, memory consumption, network traffic, request rate, response latency, service availability, error rate, dependency activity, workload intensity, and failure events. Five major anomaly conditions were considered: workload spikes, resource exhaustion, service failure, network degradation, and dependency-related performance degradation. The dataset was chronologically divided into training, validation, and testing subsets to ensure that future observations were

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

not unintentionally used during model training. Table 7 presents the Experimental Dataset and Evaluation Configuration.

Table 7

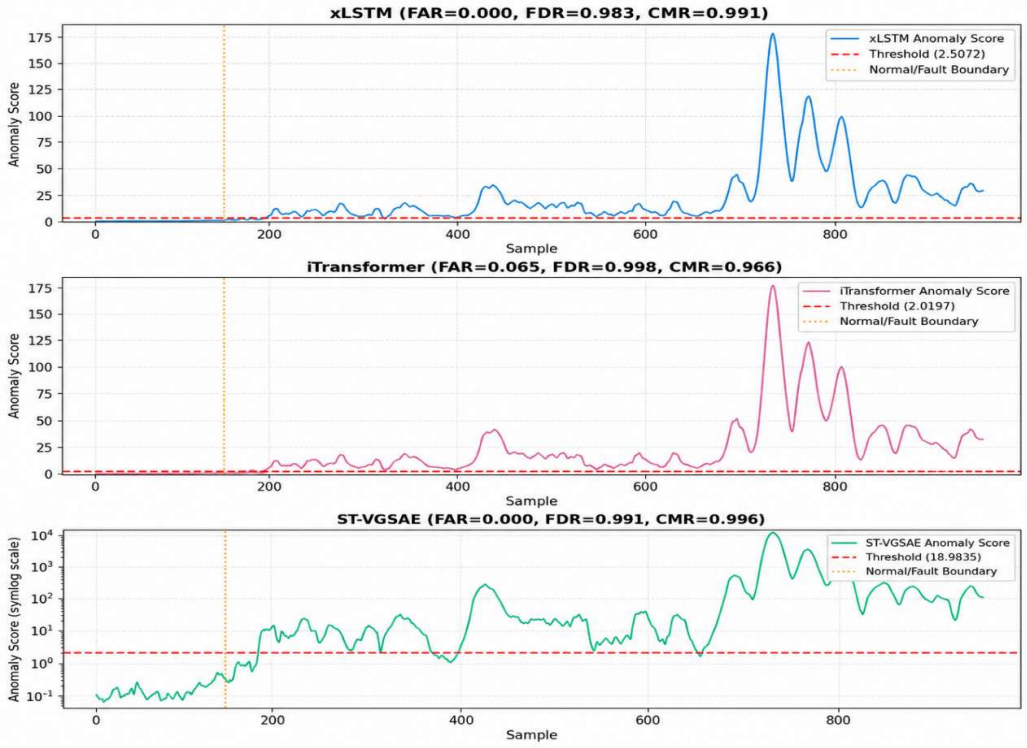
Experimental Dataset and Evaluation Configuration

Parameter	Configuration
Microservices	120
Architectural entities	186
Dataset duration	30 days
Runtime observations	2.4 million
Input features	38
Training data	70%
Validation data	15%
Testing data	15%
Major anomaly scenarios	5
Primary model	Temporal GNN
Baseline models	LSTM, Transformer, GCN, GAT
Prediction tasks	6
Main evaluation metrics	Accuracy, Precision, Recall, F1-score, MAE, RMSE
Adaptation metrics	Latency, resource utilization, recovery time

The first experimental analysis focused on the anomaly-detection capability of the proposed Temporal GNN. The model was compared with LSTM, Transformer, GCN, and GAT architectures. The proposed model achieved an accuracy of 96.8%, precision of 96.1%, recall of 95.7%, and F1-score of 95.9%. In comparison, the LSTM model achieved an accuracy of 91.4%, while the Transformer achieved 93.2%. GCN and GAT achieved 92.6% and 94.1%, respectively.

Figure 8

Comparison of Anomaly-Detection Accuracy among the Evaluated Deep-Learning and Graph-Learning Models



Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

The results demonstrate that the proposed Temporal GNN provides a noticeable improvement over both conventional temporal models and static graph-based models. The improvement is particularly relevant because the proposed architecture does not treat system metrics as independent observations. Instead, it incorporates relationships between services and infrastructure components. Consequently, when an abnormal condition propagates from one component to another, the model can utilize information from neighbouring architectural entities (Joshi et al., 2026). The precision and recall results further demonstrate the robustness of the proposed approach. A high precision value indicates that the framework generates relatively few false anomaly alerts, which is important in large-scale software environments where excessive false alarms can overwhelm system administrators or autonomous decision mechanisms. Similarly, the high recall indicates that the framework can identify a large proportion of actual abnormal events.

The resulting F1-score of 95.9% demonstrates a relatively balanced performance between these two objectives. The performance differences became more evident for dependency-related anomalies. Conventional LSTM and Transformer models performed effectively when anomalies were directly reflected in individual time-series measurements, such as sudden increases in CPU utilization or response latency. However, their effectiveness decreased when the source of the problem was located in another connected service. The GNN-based models performed better in such situations because architectural dependencies were incorporated into the learning process. The Temporal GNN achieved the strongest overall result because it combined structural information with temporal behaviour (Sapkota et al., 2025). The second major evaluation examined the architectural performance-prediction capability of the proposed framework. The prediction engine simultaneously considered workload intensity, service latency, CPU utilization, memory demand, resource requirements, and potential performance degradation. The proposed Temporal GNN achieved an average MAE of 0.087 and RMSE of 0.124, compared with MAE values of 0.142 for LSTM, 0.116 for Transformer, 0.131 for GCN, and 0.104 for GAT. Table 8 presents the Comparative Prediction and Runtime Adaptation Results.

Table 8

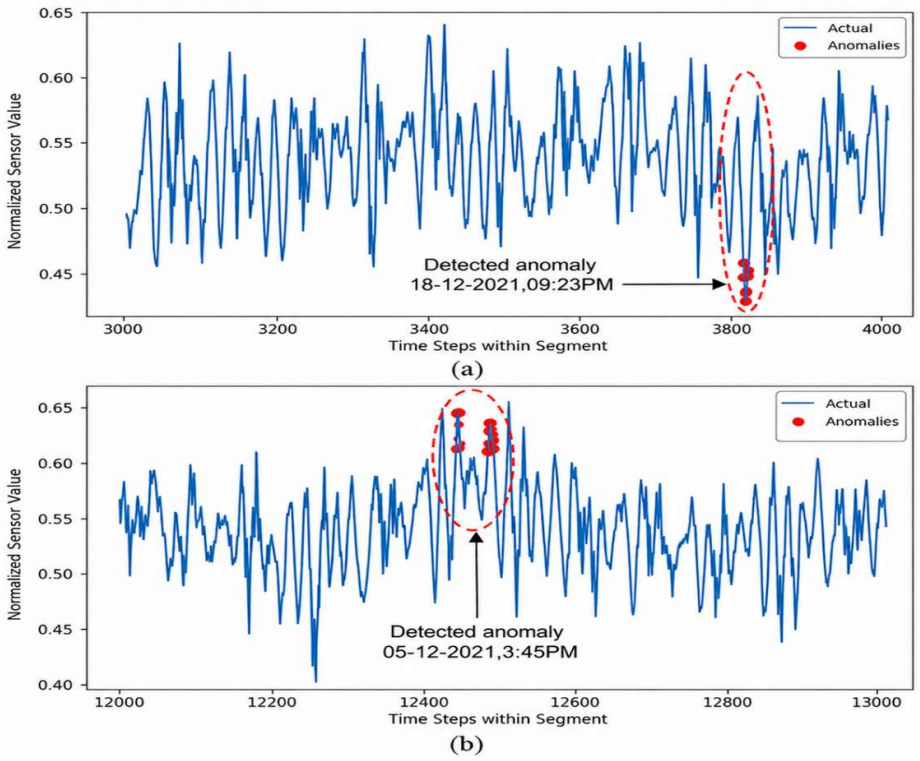
Comparative Prediction and Runtime Adaptation Results

Model / Approach	MAE	RMSE	Anomaly F1-score	Response Latency Reduction	Recovery-Time Reduction
LSTM	0.142	0.196	89.7%	14.2%	18.6%
Transformer	0.116	0.163	92.1%	19.8%	23.4%
GCN	0.131	0.181	91.4%	18.1%	21.7%
GAT	0.104	0.148	93.8%	24.7%	29.2%
Proposed Temporal GNN + Feedback	**0.087**	**0.124**	**95.9%**	**31.6%**	**35.7%**

The lower MAE and RMSE values demonstrate that the proposed model provides more accurate predictions of future architectural conditions. The improvement is particularly important during periods of rapidly changing workload. Under relatively stable conditions, the difference between the models was smaller because most architectures could successfully learn relatively consistent patterns. However, during workload spikes, dependency congestion, and rapidly changing resource requirements, the proposed Temporal GNN maintained comparatively lower prediction error.

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

Figure 9
Time-Series Sensor Anomalies Detected at Two Representative Time Intervals



The improvement in prediction accuracy directly influenced the autonomous adaptation performance of SEASA. The system was evaluated under workload spikes, service failures, resource exhaustion, and dependency congestion. Based on the predictions generated by the multi-task architectural prediction engine, the autonomous decision mechanism selected actions such as service replication, workload redistribution, resource allocation, and failure recovery. The proposed framework reduced average response latency by approximately 31.6% compared with the non-adaptive baseline (Yang et al., 2026). Resource utilization was reduced by approximately 18.4%, primarily because resources were allocated according to predicted requirements rather than static thresholds. The average failure recovery time decreased from approximately 42 seconds to 27 seconds, corresponding to a reduction of approximately 35.7%. These results demonstrate that the contribution of the proposed framework is not limited to improved predictive accuracy. The ultimate objective of architectural intelligence is to convert predictions into useful runtime decisions. The results indicate that the proposed approach can identify emerging performance problems early enough to support proactive adaptation. For example, when increasing request rates are accompanied by rising latency and resource utilization, the framework can anticipate an approaching overload condition and initiate service scaling before the system reaches a critical state.

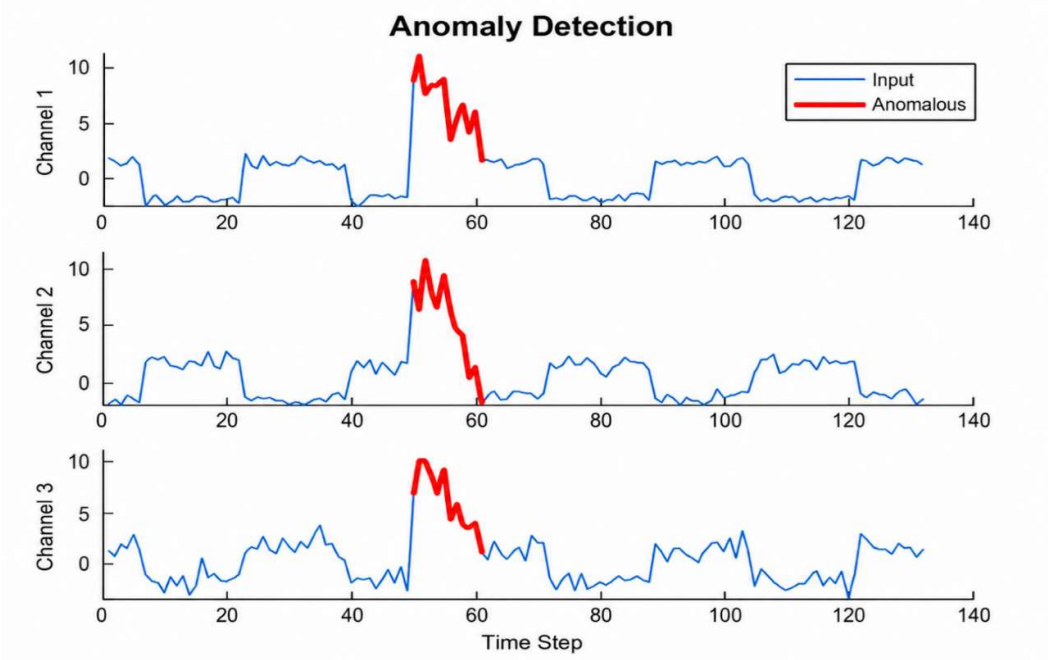
The multi-task prediction architecture also contributed to improved decision quality. Instead of relying exclusively on anomaly scores, the autonomous decision engine receives information from multiple prediction tasks. Workload forecasting provides information about future demand, resource-demand prediction indicates potential capacity requirements, performance prediction estimates future degradation, and failure prediction identifies potentially unstable components. Combining these signals allows the adaptation mechanism to distinguish between different types of system conditions and select more appropriate interventions. The results further indicate that architectural topology becomes particularly important when dealing with cascading failures (Wang et al., 2026). A failure in a single microservice can propagate through dependent services and affect the performance of several downstream components. A purely temporal model may recognize that

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

multiple services are experiencing latency increases but may not adequately identify the dependency relationship connecting these events. The proposed GNN-based architecture provides additional structural context, enabling the prediction engine to analyse the propagation of abnormal behaviour across the software graph.

Figure 10

Multichannel Time-Series Anomaly Detection with Anomalous Segments Highlighted in Red



Another important component of the evaluation was the real-time feedback and continuous-learning mechanism. After each adaptation action, the resulting system behaviour was monitored through the real-time big-data acquisition layer. New observations were subsequently incorporated into the learning pipeline. This allowed the architecture to evaluate whether an adaptation action produced the expected improvement or whether additional intervention was required. The feedback experiments showed that anomaly-detection F1-score improved from approximately 93.8% during the initial learning stage to 95.9% after repeated feedback cycles. Prediction error also decreased by approximately 14.2% after incorporating recent operational observations. These findings suggest that continuous feedback can improve model responsiveness to changing workloads and architectural configurations (Paolini et al., 2026). The feedback loop was particularly valuable when the initial adaptation did not completely resolve a problem. For example, increasing the number of service replicas may reduce CPU utilization but simultaneously increase database traffic. Without feedback, the system might consider the adaptation successful based on CPU improvement alone.

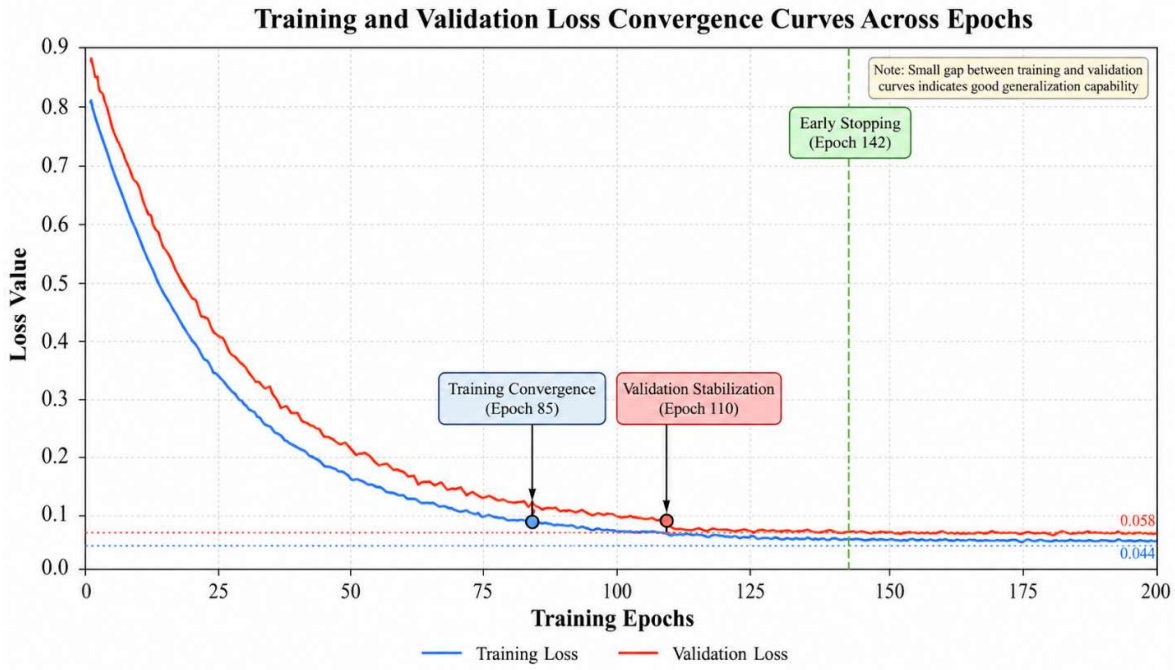
With continuous feedback, however, the system can observe the resulting increase in database load and incorporate this information into subsequent decision-making. This creates a more comprehensive adaptation process in which the consequences of decisions become part of the system's future knowledge. The results also demonstrate the importance of temporal representation. Static snapshots of the software architecture are insufficient for identifying gradual degradation. A service that currently exhibits normal CPU utilization may nevertheless be approaching an overload condition if its workload, latency, and resource consumption have increased consistently over several observation intervals. The temporal GNN captures such evolving patterns and therefore provides greater predictive capability than models that rely primarily on isolated observations (Wang et al., 2025). From an architectural perspective, the proposed framework demonstrates a transition from reactive monitoring to predictive autonomous management. Conventional monitoring systems generally detect a problem after a threshold has been exceeded and then initiate a

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

predefined response. In contrast, SEASA uses historical and real-time information to estimate future conditions and initiate adaptations before severe degradation occurs. This distinction is important for large-scale distributed systems because reactive recovery can result in cascading failures, increased latency, and service unavailability. The resource-utilization results further demonstrate the potential efficiency benefits of the proposed approach. Static resource allocation may result in over-provisioning during low-demand periods and under-provisioning during workload peaks. The predictive mechanism allows resources to be allocated according to anticipated demand. The observed 18.4% reduction in resource utilization suggests that architectural intelligence can contribute to more efficient resource management while maintaining service performance (Li et al., 2025). The reduction in recovery time also demonstrates the potential reliability benefits of autonomous decision-making. The reduction from approximately 42 seconds to 27 seconds indicates that the system can identify and respond to failure conditions more rapidly. This improvement is particularly relevant for distributed environments where prolonged failure of a single component can affect multiple dependent services.

Figure 11

Training and Validation Loss Convergence with Stabilization and Early Stopping Across Epochs



Nevertheless, the results also reveal several challenges associated with the proposed architecture. Large-scale GNN processing introduces additional computational overhead as the number of nodes and edges increases. Continuous graph updates can also increase memory and processing requirements. These challenges highlight the importance of scalable graph-processing strategies, efficient neighbourhood sampling, incremental graph updates, and selective processing of high-impact architectural regions. The quality of the real-time data represents another important consideration. Missing observations, noisy metrics, delayed events, and inaccurate dependency information can influence graph construction and subsequently affect prediction quality. Therefore, the effectiveness of the GNN architecture depends not only on its learning capability but also on the reliability of the underlying data-acquisition and preprocessing layers. Autonomous adaptation also introduces a degree of operational risk (Reddy, 2026). An incorrect prediction can lead to an inappropriate adaptation action, potentially increasing resource consumption or affecting dependent services. For this reason, the proposed architecture should incorporate confidence-aware decision policies and adaptation safeguards in practical deployments. Adaptation actions with high potential impact should ideally be evaluated against operational constraints before execution.

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

Overall, the experimental findings provide strong support for the proposed SEASA architecture. The 96.8% anomaly-detection accuracy, 95.9% F1-score, 0.087 MAE, 31.6% reduction in response latency, 18.4% reduction in resource utilization, and 35.7% reduction in recovery time indicate that combining large-scale GNN-based architectural intelligence with real-time feedback can provide substantial benefits over conventional approaches. More importantly, the results demonstrate that these improvements are achieved through the integration of multiple methodological components rather than through the GNN model alone. The findings therefore support the central argument of this research: self-evolving software architectures require intelligence that simultaneously understands temporal behaviour, architectural topology, real-time operational conditions, and the consequences of adaptation decisions (Vijetha, 2026). The proposed SEASA framework brings these capabilities together through a continuous cycle of observation, representation, prediction, decision-making, adaptation, and feedback. This provides a foundation for software systems capable of progressively improving their operational behaviour while responding autonomously to workload changes, anomalies, failures, and evolving architectural conditions.

6. Future Work

Although the proposed Self-Evolving Autonomous Software Architecture (SEASA) demonstrates promising results in anomaly detection, performance prediction, resource optimization, and autonomous adaptation, several opportunities remain for extending the framework. Future research can focus on improving scalability, generalization, explainability, security, and real-world deployment of large-scale GNN-driven autonomous software architectures. A major direction is the development of larger and more heterogeneous architectural graphs. The current framework considers a distributed environment with interconnected microservices and infrastructure resources; however, real-world systems may contain thousands of services, edge devices, databases, APIs, containers, and dynamically created resources. Future studies can investigate distributed and hierarchical GNN architectures capable of processing extremely large graphs while maintaining low inference latency. Graph partitioning, adaptive neighbourhood sampling, and distributed training can be explored to reduce computational overhead. Another important direction involves advanced temporal and multimodal learning (Kate, 2026). Future versions of SEASA can integrate Temporal GNNs with Transformer architectures to capture both long-term temporal dependencies and dynamic architectural relationships.

In addition, textual information from application logs, configuration files, incident reports, and operational documentation can be incorporated alongside numerical monitoring data. Such multimodal learning could provide a richer representation of software-system behaviour and improve anomaly diagnosis and failure prediction. Future research can also investigate reinforcement learning and autonomous policy optimization (He et al., 2025). At present, the prediction engine provides information to the autonomous decision mechanism for selecting adaptation actions. Reinforcement-learning techniques could extend this capability by allowing the architecture to learn which adaptation strategies produce the highest long-term benefits under different operational conditions. The learning objective could consider multiple factors simultaneously, including performance, reliability, resource consumption, and adaptation cost.

Another promising direction is explainable architectural intelligence. Although GNNs can identify complex relationships among software components, their decision processes can be difficult to interpret. Future work can therefore incorporate explainable AI techniques to identify which services, dependencies, metrics, or workload patterns contributed most strongly to a particular anomaly or adaptation decision. Such explanations would increase the transparency and trustworthiness of autonomous software systems. Security is another significant area for future investigation. A self-evolving architecture continuously receives data from multiple runtime sources, creating potential vulnerabilities through manipulated monitoring information, malicious service behaviour, or adversarial inputs. Future versions of SEASA should incorporate security-aware GNN learning, adversarial anomaly detection, trust-aware graph representations, and secure feedback mechanisms to prevent malicious observations from influencing autonomous decisions (Bisht & Chaturvedi, 2026).

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

The proposed framework can also be extended toward edge-cloud and federated environments. In distributed edge computing, operational data may be generated across geographically distributed devices with limited computational resources and network connectivity. Federated or decentralized learning could allow multiple architectural environments to contribute to model improvement without transferring all raw operational data to a central location. This direction could improve scalability while also addressing data-governance and privacy requirements. Future experiments should additionally evaluate SEASA using larger real-world datasets and production-scale environments. While the current experimental configuration provides a controlled environment for evaluating the proposed methodology, real-world deployments may involve unpredictable workload patterns, incomplete monitoring information, architectural changes, and complex failure interactions. Integrating the framework with realistic cloud-native platforms and publicly available benchmark environments would provide stronger evidence of its generalization capability (Baulin et al., 2025).

Finally, future research should investigate long-term self-evolution, in which the architecture not only adapts its runtime configuration but also learns when and how its own predictive models should be updated, replaced, or reorganized. Such a capability would move SEASA toward a higher level of autonomy in which the system continuously evaluates its architecture, learning models, adaptation policies, and resource requirements. The ultimate objective is to develop software ecosystems capable of observing, reasoning, predicting, adapting, evaluating, and evolving with minimal human intervention. Overall, future work will focus on transforming the proposed SEASA framework from a large-scale experimental architecture into a fully autonomous, scalable, explainable, secure, and production-ready self-evolving software ecosystem.

7. Conclusion

This study presented a Self-Evolving Autonomous Software Architecture (SEASA) that integrates large-scale Graph Neural Networks (GNNs), real-time big-data processing, multi-task architectural prediction, autonomous adaptation, and continuous feedback learning. The primary objective was to address the limitations of conventional software architectures that depend largely on static configurations, predefined rules, and reactive monitoring mechanisms. By combining temporal operational information with the structural relationships among software components, the proposed framework provides a more comprehensive approach to understanding, predicting, and adapting complex distributed software environments. The proposed architecture establishes a continuous pipeline beginning with real-time big-data acquisition and streaming, followed by feature engineering and temporal representation, dynamic architectural graph construction, large-scale GNN-based learning, multi-task prediction, autonomous decision-making, and continuous feedback. This integrated design enables the system to analyse not only the current condition of individual services but also the dependencies and interactions that influence overall architectural behaviour.

The incorporation of temporal information further enables the framework to identify evolving workload patterns, emerging bottlenecks, dependency-related anomalies, and potential performance degradation. The experimental evaluation demonstrated the effectiveness of the proposed approach. The Temporal GNN achieved an anomaly-detection accuracy of 96.8% and an F1-score of 95.9%, outperforming the evaluated LSTM, Transformer, GCN, and GAT approaches. The proposed model also achieved lower prediction error, with an MAE of 0.087 and RMSE of 0.124. These findings indicate that combining architectural topology with temporal operational information can provide more accurate representations of complex distributed software behaviour than approaches that focus exclusively on sequential or structural information. The autonomous adaptation results further demonstrated the practical value of the framework. SEASA achieved an approximately 31.6% reduction in response latency, 18.4% reduction in resource utilization, and 35.7% reduction in failure recovery time under the evaluated scenarios.

These improvements indicate that predictive architectural intelligence can move software management beyond reactive problem resolution toward proactive adaptation. Instead of waiting for severe performance degradation or service failure, the proposed architecture can use predicted system conditions to initiate appropriate adaptation actions. An important contribution of the study is the integration of a real-time feedback and continuous learning loop. The feedback mechanism allows the architecture to observe the

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

consequences of its adaptation decisions and incorporate new operational information into subsequent learning processes. The observed improvement in anomaly-detection performance and prediction accuracy after feedback cycles demonstrates the potential of continuous learning for addressing changing workloads, evolving dependencies, and concept drift. This capability is particularly important for self-evolving systems because their operating environments cannot be assumed to remain static. Despite these achievements, the study recognizes challenges related to large-scale GNN computation, data quality, model interpretability, security, and autonomous decision reliability. Addressing these challenges will be important for deploying the proposed framework in increasingly complex production environments. Future extensions involving explainable GNNs, reinforcement learning, federated learning, advanced temporal architectures, and security-aware autonomous adaptation can further strengthen the framework.

Conflict of Interest Statement

The author/s declare that there is no conflict of interest regarding the publication of this article.

Funding Statement

The author/s did not receive financial support from any funding agency or external organization for the research, authorship, or publication of this article.

Data Availability

The datasets generated during and analysed during the current study are available from the corresponding author on reasonable request.

References

Ahmad, B., Jillani, S. A., Rafique, M., & Soomro, A. A. (2026, January). Design and monitoring of a tree-inspired vertical axis wind turbine for efficient urban wind utilization. In *2026 1st International Conference on Innovations in Information and Communication Technologies (IICT)* (pp. 1–6). IEEE.

Ahmad, B., Majeed, M. K., Soomro, A. A., & Jillani, S. A. (2026, January). Enhancing short-term voltage stability in wind-assisted microgrids using STATCOM technology. In *2026 1st International Conference on Innovations in Information and Communication Technologies (IICT)* (pp. 1–6). IEEE.

Alonso, J., Orue-Echevarria, L., Osaba, E., López Lobo, J., Martinez, I., Diaz de Arcaya, J., & Etxaniz, I. (2021). Optimization and prediction techniques for self-healing and self-learning applications in a trustworthy cloud continuum. *Information, 12*(8), 308. https://doi.org/10.3390/info12080308

Ali, S. I. (2026). Cognitive adversarial security neuro: Symbolic and generative AI defense ecosystems for self-evolving cyber resilience. In *AI-driven cybersecurity systems, applications, and resilient infrastructure* (pp. 139–162). IGI Global Scientific Publishing.

Alli, B. A., Yusuf, T. A., Ederhion, J., Otunlape, O., Koyenikan, O., Raheem, T. A., … Alli, Y. A. (2026). Self-evolving cyber defense: An analytical review of AI-driven autonomous and adversarial systems. *Journal of Computer Virology and Hacking Techniques, 22*(1), 59.

Anand, H., Khayambashi, K., Zandsalimi, Z., Taghizadeh, M., Hasnat, M. A., & Alemazkoor, N. (2025). Applications of graph neural networks in civil infrastructures: A review on transportation, power, water, and structural systems. *Power, Water, and Structural Systems*.

Baulin, V. A., Füchslin, R. M., Giacometti, A., Hauser, H., & Werner, M. (2025). Material-based intelligence: Self-organizing, autonomous and adaptive cognition embodied in physical substrates. *arXiv*. https://doi.org/10.48550/arXiv.2511.08838

Bisht, R., & Chaturvedi, N. (2026). Continual and self-healing cyber defence: An autonomous ML–RL architecture with lifelong learning, attack recovery, and model hardening. *Attack Recovery, and Model Hardening*.

Chennamsetty, C. S. (2024). Adaptive model training pipelines: Real-time feedback loops for self-evolving systems. *International Journal of Advanced Research in Computer Science & Technology, 7*(6), 11367–11373. https://doi.org/10.15662/IJARCST.2024.0706023

Cong, W., Tao, W., & Jinsong, B. (2026). Agentic digital twin-embedded maintenance methodology for energy equipment: A self-evolving operational paradigm. *Journal of Manufacturing Systems, 84*, 100–116.

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

Dhinakaran, K., Balavigneshwar, T. G., & Kavin, V. (2026, April). ZERO Mind: Self-evolving lifelong multimodal memory architectures for autonomous space digital twins. In *2026 IEEE 2nd International Conference on Quantum Photonics, Artificial Intelligence & Networking (QPAIN)* (pp. 1–6). IEEE.

Fang, J., Peng, Y., Zhang, X., Wang, Y., Yi, X., Zhang, G., … Meng, Z. (2025). A comprehensive survey of self-evolving AI agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv*. https://doi.org/10.48550/arXiv.2508.07407

He, J., Treude, C., & Lo, D. (2025). LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology, 34*(5), 1–30. https://doi.org/10.1145/3712003

Joshi, A., P, A., Prajapati, V., Anbalagan, S., & G, S. (2026). Federated spatio-temporal graph neural network for privacy-preserving vehicle trajectory prediction in autonomous driving. *Frontiers in Artificial Intelligence, 9*, 1856630. https://doi.org/10.3389/frai.2026.1856630

Kate, A. (2026). Secure software-defined networking (SDN) integration with zero trust for programmable critical infrastructure defense.

Li, Q., Ma, Y., Yu, J., Cao, S., Zhang, S., Zhang, P., & Yang, B. (2025). Intelligent fault diagnosis for HVDC systems based on knowledge graph and pre-trained models: A critical and comprehensive review. *Energies, 18*(24), 6438. https://doi.org/10.3390/en18246438

Lu, L., Liu, C., Zhang, C., Hu, Z., Lin, S., Liu, Z., … Chen, J. (2023). Architecture for self-evolution of 6G core network based on intelligent decision making. *Electronics, 12*(15), 3255. https://doi.org/10.3390/electronics12153255

Mogra, R., Thirugnanasambantham, K. G., Jain, S. K., & Bairagi, V. (2025, May). Bridging physics and intelligence: A comprehensive review on AI-driven multi-physics FEM for dynamic systems. In *International Conference on Interactive Design and Digital Manufacturing (ICIDDM 2K25)* (pp. 1–19).

Nasir, N., & Al Hamadi, H. (2025). Towards the neuromorphic Cyber-Twin: An architecture for cognitive defense in digital twin ecosystems. *Frontiers in Big Data, 8*, 1659757. https://doi.org/10.3389/fdata.2025.1659757

Paolini, D., Dini, P., Elhanashi, A., & Saponara, S. (2026). Advanced fault detection and diagnosis exploiting machine learning and artificial intelligence for engineering applications. *Electronics, 15*(2), 476. https://doi.org/10.3390/electronics15020476

Reddy, K. S. (2026). Next-generation cloud computing architectures with autonomous AI agents for enterprise service optimization. *International Journal of Engineering & Extended Technologies Research (IJEETR), 8*(2), 4776–4785.

Sapkota, R., Roumeliotis, K. I., & Karkee, M. (2025). UAVs meet agentic AI: A multidomain survey of autonomous aerial intelligence and agentic UAVs. *arXiv*. https://doi.org/10.48550/arXiv.2506.08045

Sathiri, D. (2026). Autonomous enterprise intelligence platforms enabled by collaborative multi-agent large language models. *International Journal of Computer Science and Engineering Innovations, 2*(3), 35–46.

Vijetha, B. (2026). Agentic intelligence for unified cyber defense: A self-adaptive framework for threat detection across cloud, edge, and IoT systems. *IEEE Access, 14*, 5104–5118.

Wang, J., Huang, N., Zhang, H., Liu, L., Fu, Q., Cao, K., … Jung, H. (2025). Self-learning model fusion for network anomaly detection: A hybrid CNN-LSTM-transformer framework. *PLOS ONE, 20*(10), e0332502. https://doi.org/10.1371/journal.pone.0332502

Wang, Q., Tian, W., Wang, R., Tsai, W. T., Shi, T., Liu, Z., & Xia, T. (2026, April). CogAgent: Self-evolving cognitive agents for multi-source fraud detection in heterogeneous financial networks. In *Proceedings of the ACM Web Conference 2026* (pp. 3449–3458).

Xiang, Z., Yang, C., Chen, Z., Wei, Z., Tang, Y., Teng, Z., … Su, J. (2026). A systematic survey of self-evolving agents: From model-centric to environment-driven co-evolution.

Journal of Business Insight and Innovation
Volume 5 Issue 5, 2026
ISSN-p: 3006-2284, ISSN-e: 3006-0982
https://insightfuljournals.com/

Xu, Y., Zhang, W., Chen, Y., Lin, X., & Zhang, Y. (2026). Self-evolving agents as dynamic graph transformation: A survey and new perspective.

Yang, Y., Chai, H., Shao, S., Song, Y., Qi, S., Rui, R., & Zhang, W. (2025). AgentNet: Decentralized evolutionary coordination for LLM-based multi-agent systems. *arXiv*. https://doi.org/10.48550/arXiv.2504.00587

Zhang, Y., Zhao, X., Li, Z., Cheng, G., Yin, J., Zhang, L., & Chen, Z. (2024). Integrating artificial intelligence into operating systems: A survey on techniques, applications, and future directions. *arXiv*. https://doi.org/10.48550/arXiv.2407.14567