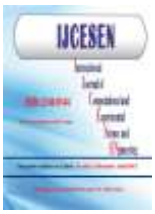
International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 12-No.2 (2026) pp. 11-18
http://www.ijcesen.com



ISSN: 2149-9144

Research Article

A Policy-Driven Adaptive Resilience Framework for Spring Boot Microservices and Angular Frontends

Sreelatha Pasuparthi*

KSRM College of Engineering, India
* **Corresponding Author Email:** sreelathapasuparthi@gmail.com - **ORCID:** 0000-0002-0047-7598

Article Info:

DOI: 10.22399/ijcesen.5110
Received : 05 February 2026
Revised : 29 March 2026
Accepted : 30 March 2026

Keywords

Policy-Driven Resilience,
Microservice Fault Tolerance,
Runtime Contract Enforcement,
Adaptive Policy Management,
Full-Stack Observability

Abstract:

Microservice architectures and rich single-page application (SPA) frontends have become the standard case for enterprise software, yet resilience on both the microservices backend and frontend remains an unresolved engineering problem. The article presents a policy-driven adaptive resilience framework based on runtime contract enforcement and dynamic policy management for Spring Boot microservices and Angular frontends. The architecture consists of three components: first, a central policy engine for all resilience contracts in different versions. Second, a Runtime Contract Enforcer, which transparently intercepts service calls at the back end and the front end. Third, a Monitoring and Feedback Loop component for condition- and knowledge-based policy adaptation. Providing hot-reloadable policy propagation, contract-driven fault tolerance, and full-stack coverage without redeploying services sets apart this article with respect to prior work in static circuit-breaking, Resilience4j, and Istio/Envoy. Businesses evaluate the framework under various common failure scenarios and show improvements to mean time to recover, fault rate under failure, and latency to policy updates. The article concludes that the framework is a scalable and operationally agile solution for dynamic enterprise deployments.

1. Introduction

The use of microservice architectures and SPA front-ends in enterprise software systems has introduced many changes to key operational characteristics. Microservices cut large monolithic applications into fine-grained, independently deployable services that can be independently run over a network. These approaches introduce benefits such as scalability and a number of additional failure modes, including those introduced by network instability, the number of services, and the distribution of state management [1]. Spring Boot is the most widely used Java microservice framework for production, while Angular is the most widely used framework for enterprise web front-ends. The two frameworks, when paired, form the technical foundation for most modern enterprise applications across industries such as financial services, healthcare, logistics, advertising, and e-commerce.

While microservices architectures are now mature, their nature notably alters the resilience challenges of their predecessors. By replacing in-process method calls with network calls, microservices may be more sensitive to latency spikes, partial failures, and cascading failures. In fact, Dragoni et al. [1] argue that the microservice-based architecture leads to a fine-grained approach to service decomposition and an increase in the operational complexity to isolate failure and manage service dependencies. This raises the problem of how to develop frontend applications with a graceful degradation when backend services are not available. The interaction of frontend and backend service failure modes leads to complex operational scenarios that are only partially addressed by resilience patterns.

Static circuit breakers, retry policies, and fallback logic are compiled into binaries and fixed at the application logic level. Changing these policies requires changing code, re-running tests, and redeploying services. This can hamper the speed of change and lead to system fragility. In cases of high business agility or low tolerable latency for incident handling, these structural limitations lead to prolonged or poor user experiences in terms of the period of an outage. Intrinsic within the current generation of resilience libraries is the inability to

dynamically change the resilience behavior of a system without a dependency on a deployment pipeline.

To address these limitations, the article proposes a policy-driven adaptive resilience framework, which is based on three principles: (1) abstraction of resilience contracts as first-class artifacts in a centralized Policy Engine with versioning support, (2) transparent interception and control of service invocations through a Runtime Contract Enforcer over Spring Boot and Angular layers. Third, the addition of a Monitoring and Feedback Loop that enables adaptive policy changes in case of triggering conditions. The remainder of this article is organized as follows: it discusses related work, describes the architecture and implementation, presents experiments, and concludes with implications for enterprise resilience engineering.

2. Related Work

Resilience frameworks for distributed microservices are well-established, but major difficulties remain in runtime adaptability, business-level contract enforcement, and the full-stack support for both back-end and front-end services.

Netflix Hystrix was one of the first successful circuit breaker patterns to be implemented in the JVM ecosystem for microservices. Key patterns in the Hystrix library are documented in the GitHub repository [2]: failure isolation via thread pooling and semaphore bulkheading, circuit breaker state management, and synchronous fallback execution. As Hystrix has thus been put into maintenance mode without any further feature development, its policies can only be configured statically at application start-up, making it less applicable to continuously operated production applications where threshold and fallback behaviors need to be modified at runtime without redeployment.

Resilience4j is the recommended standard library to implement resilience patterns in Java applications. It is lightweight, modular, and distributed. The documentation [3] describes a composable set of decorators for circuit breaking, rate limiting, retry, bulkhead, and time limiting with an emphasis on functional programming and low dependencies. Although more modular and testable than Hystrix, Resilience4j's policies have to be set up at application startup, and modifying any retry limit or circuit breaker threshold forces a redeployment of the application, causing a lengthy delay between the incident and application of the policy in certain failure-prone applications. This is particularly costly for applications requiring high availability.Liveness, readiness and startup probes are Kubernetes health checks that provide lifecycle management at the container level, allowing the orchestration layer to restart unresponsive containers and avoid sending traffic to unready containers [4]. Since this coarse-grained recovery strategy is done at the infrastructure level, it cannot express application-level fallbacks, define retry semantics at the service call level, or specify business-level policy conditions and trade-offs.

Istio and Envoy promote the service mesh pattern, where resilience logic is moved from the service to a network-layer proxy sidecar. Istio documentation [5] describes configurable retries, timeouts, circuit breaking, and traffic shifting via the control plane. However, the mesh is unaware of application-level contracts and business semantics. The applications running in the browser on the frontend cannot be part of the mesh boundary, and Angular clients in particular cannot have the same resilience guarantee as other services in the mesh. Spring Cloud Circuit Breaker [6] provides a vendor-neutral abstraction over these backends, but it does not solve their limitations regarding static configuration, nor provide an out-of-the-box frontend integration, nor allow modifying policies at runtime. The work addresses these issues with a full-stack unified and runtime-adaptive architecture.

3. Proposed Approach

3.1. Architecture Overview

The proposed framework consists of three coordinated parts that make up a closed-loop resilience control plane: the Policy Engine, the Runtime Contract Enforcer, and the Monitoring and Feedback Loop.

The Policy Engine is a separate Spring Boot service that has a versioned REST API for making, getting, updating, and deleting policies. Policies are modeled as structured contracts—named, versioned documents associating a target service and endpoint with a set of conditional resilience actions. The engine persists contracts in a distributed database and maintains a version history to support rollback and audit. A web-based management UI enables operators to author and publish policies without direct API interaction, and hot-reload semantics ensure that policy updates propagate to consuming services in near real time without requiring process restarts.

The Runtime Contract Enforcer is a lightweight, embeddable library that intercepts service calls at runtime and applies the active policy contract for the targeted endpoint. For Spring Boot services, enforcement is implemented via Spring AOP, weaving resilience behavior into REST and gRPC

client invocations transparently and without modification to business logic. For Angular applications, a globally registered HTTP interceptor intercepts all outbound requests and applies frontend-specific resilience behaviors, including retry, fallback UI rendering, and offline action queuing. The enforcer maintains a local cache of policies fetched from the Policy Engine, refreshing via push notification or scheduled polling.

The Monitoring and Feedback Loop treats observability as a first-class concern. Each enforcer instance exposes various metrics through Micrometer (in Prometheus scrape format), including the number of retries, changes to the circuit breaker state, fallbacks and distribution of latencies for each endpoint. System health and policy performance are displayed on Grafana [8] dashboards. In case an SLO threshold is exceeded, either the cluster will call the Policy Engine's update API or a corresponding alert will be sent through to the appropriate alerting mechanism set up for this workload. The Drools rule engine [9] can be used to implement declarative business logic, which allows automatic mutation decisions to be made based on certain metric patterns (closed-loop, self-healing).

3.1.1. Policy Contract Schema Example

```
policyId: payment-retry-policy
service: payment-service
endpoint: /api/payments
conditions:
  - error Type: TimeoutException
    action:
      retry:
        maxAttempts: 3
        backoff: 2000 ms
      fallback:
        response: { "status": "pending" }
  - errorType: CircuitBreaker Open
    action:
      fallback:
        response: { "status": "unavailable" }
```

3.1.2. Policy Distribution

Policy distribution is a combination of push and pull. The Policy Engine sends change notifications via webhooks to registered enforcer instances. Upon each policy update, both enforcer instances refresh their cache. In other cases where delivery is not guaranteed, enforcers also poll for Policy changes at user-configurable intervals. Local caching allows enforcement to continue even in case of transient Policy Engine failures.

3.2. Key Features

Dynamic policy updates allow resilience policies to be changed at runtime without the need to change the application code or necessarily trigger the deployment pipeline by using the Policy Engine's API. Contract-oriented design provides a model for externalizing all fault tolerance and resilience capability into external, named, versioned artifacts. Full-stack coverage enables resilience policy enforcement in the Angular frontend. The policy infrastructure is shared with the backend. Each enforcer instance exposes its operational metrics by service, endpoint, policy version, and action type to enable SLO-based fine-grained alerting. The action model also supports extension points to create ServiceNow incidents, publish failure events to a message broker, or invoke secondary fallback services to integrate into existing operational toolchains.

4. Implementation

4.1 Spring Boot Integration

The Spring Boot services are supported by a starter library that will automatically configure everything once added to the application's dependency management. On application startup, the starter registers the Runtime Contract Enforcer as an AOP proxying aspect. The Policy Agent also connects to the Policy Engine using application properties, sets up a local policy cache, and uses an AOP aspect to intercept calls to explicitly annotated service client methods or, optionally, all outbound HTTP/grpc calls using a global pointcut. On each call, the enforcer determines which is the policy applicable to the endpoint, which conditions are matching, and which resilience actions that policy prescribes should be executed.

The retry logic is configurable and supports linear, exponential, and fixed backoff strategies. The use of jittering is optional to avoid the thundering herd problem that can occur if many endpoints receive failures at the same time. Circuit breaker state is maintained per endpoint and transitions between closed, open and half-open states based on policy. After sending a policy update notification over the webhook, the whole policy cache is replaced with the new policy by acquiring a read-write lock to avoid race conditions during the hot-reload process. Micrometer is a library that exports metrics to a predefined Prometheus registry [7], and predefined Grafana dashboards as JSON templates are available [8]. Service discovery integration with Eureka and Consul enables dynamic endpoint resolution in environments where service instances are transient by nature.

4.2 Angular Integration

Angular integration is provided through an NgModule. When imported to the root module of the app, the module installs a global HTTP interceptor and initializes the frontend policy cache. The interceptor applies frontend-specific resilience behaviors to all HttpClient requests. For retries, RxJS `retryWhen` operators are used for each retry action, using the delay and count specified by the policy. For fallback actions, the interceptor can return an in-memory static response or call a secondary endpoint, and offline queuing is provided for write operations. If a backend endpoint becomes unavailable, the interceptor serializes all writes to an in-memory queue and replays them once the endpoint becomes available. The injectable notification service can be used to provide resilience feedback to users. Via the injectable notification service, consumers can override which notification actions are taken.

4.3. Policy Example

```
service: payment-service
endpoint: /api/payments
retry:
  maxAttempts: 3
  backoff: 2000 ms
fallback:
  response: { "status": "pending" }
```

4.4 Monitoring and Feedback

Prometheus scrape targets are configured using the Micrometer Prometheus registry [7]. Out-of-the-box Grafana dashboard templates [8] are available for monitoring error rates over time, retry heatmaps, circuit breaker states, and mean time to recovery. Default alert rules are defined in the Prometheus Alertmanager and shipped with the framework. Based on the firing alert, the Drools rules engine [9] determines if policy adaptation is needed and which mutation to apply. Every automated action taken by the platform is logged as a versioned policy event in the Policy Engine audit trail for complete traceability.

5. Experimental Results

5.1 Setup

The experimental setup used to compare the baseline was 10 Spring Boot microservices (service providers) and 1 Angular frontend application using Amazon Elastic Kubernetes Service (EKS) with an Istio service mesh [5]. The services represented a synthetic e-commerce system of catalog, inventory, pricing, payment, notification, authentication, order management, shipping, reporting, and API gateway services. At least 2 replicas were deployed on different availability zones (AZs). Kubernetes readiness probes and liveness probes [4] were enabled in all services in all configurations to control the lifecycle of the containers.

Businesses perform five failures on the setup to simulate fault injections: network partition isolating the payment service , crash of the inventory service , introduction of 100% latency to the order management service, cascading failures of dependent services from the notification service, and disconnection of the frontend from the backend services to simulate network loss. This article repeats the evaluations under the same load profiles for static circuit breakers, Resilience4j [3], Istio/Envoy mesh [5], and our proposed resilience framework.

5.2. Results

The proposed framework achieved the lowest MTTR across all tested failure scenarios, attributable primarily to near-instantaneous policy propagation via the hot-reload mechanism—a qualitative shift compared to the restart-or-redeploy requirement of Resilience4j [3] and static circuit breaker approaches. Error rate under fault conditions was reduced compared to all baseline approaches, reflecting the combined effect of dynamic retry orchestration and aggressive fallback application. The policy update latency of the proposed framework, enabled by push-based webhook distribution, represents a substantive operational advantage over the configuration reload cycle of Istio/Envoy [5] and the full redeploy required by static approaches.

5.2.1. Scenario Analysis

In the case of the network partition, with the error rate detected by the enforcer, the circuit breaker was opened and the fallback value that was previously defined was returned to the user, while preventing any cascading failure from reaching upstream services. In the case of the inventory service failure, the circuit breaker was opened after the retries were exhausted. The system's downtime was less than five seconds and was repeatable. The Angular interceptor switched to offline mode in case of a backend cut-off, storing the state of an incomplete form in memory and also queuing write requests to be processed upon re-establishment of a connection with the backend.

5.2.2. Visualizations

Grafana dashboards [8] configured for the proposed framework provided real-time visualization of the resilience control plane during each fault scenario. Error rate panels indicate a step-down characteristic to hot-reload of policy, confirming nearly instantaneous propagation of policy to edge devices. Update latency histograms confirmed that the substantial majority of updates were enforced within three seconds of submission, with tail latency concentrated in instances where webhook delivery experienced transient delay and the pull-based fallback mechanism was engaged.

6. Comparison

The comparison reveals several structural differentiators that distinguish the proposed framework from existing solutions across the key dimensions of adaptability, coverage, and observability. Runtime policy update—the ability to modify resilience behavior in a live system without restart or redeployment—is absent in static circuit breaker approaches and only partially supported by Resilience4j [3], whose documentation makes clear that configuration changes take effect only after application restart. Istio/Envoy [5] supports configuration reload without service restart, but this capability is confined to the network proxy layer and does not extend to application-level business logic or frontend clients.

The contract-oriented style of resilience specification, whereby resilience requirements are specified as named, versioned artifacts decoupled from application code, is a novel capability not present in any of the baseline frameworks evaluated in this work. Newman [10] advocates explicit service contracts and interface governance in microservice architectures as a means of managing inter-service dependencies and reducing coupling. The proposed framework extends this to resilience behavior with fault tolerance policies treated as governed contracts rather than implementation details.

None of the baseline approaches provides full-stack coverage, including both backend services and Angular frontend clients. Hystrix [2] and Resilience4j [3] operate exclusively within the JVM process boundary; Istio/Envoy [5] operates at the network layer; none address the browser-resident frontend. The The Angular HTTP interceptor in the proposed framework bridges this gap by governing frontend resilience behaviors under the same policy infrastructure as backend services.The monitoring feedback loop, enabling condition-triggered automated policy adaptation, is supported only partially by Istio/Envoy through traffic management policies and is absent in all other evaluated approaches. The proposed framework's integration of Prometheus [7] metrics, Grafana [8] dashboards, and a Drools-based rules engine [9] creates a closed-loop control plane in which metric-driven conditions can trigger policy mutations without operator intervention. Business policy support—the capacity to encode domain-specific conditions into resilience contracts—and the high extensibility and observability ratings of the proposed framework reflect its design as a platform for operational governance rather than a fixed-function resilience library.

7. Discussion

The policy-driven adaptive resilience framework fills an important gap in enterprise resilience building as it provides organizations with the operational flexibility to react to failures that is not possible with static, code-embedded resilience mechanisms, based on the principle that resilience is an operational property that should be controlled with policy mechanisms, not a feature of an application. Such separation of responsibility reduces cognitive load on development teams and enables incidents to be diagnosed more quickly and allows the platform engineering teams to own and evolve the resilience posture without being interrupted by the cadence of feature development. In the world of microservice governance, for example, Newman [10] argues that changing behavior across service boundaries without coordinating deployments is an enabler of organizational agility at scale [11].

Therefore, the full-stack approach is especially important for enterprise applications, where degraded performance in the front end is the leading cause of user dissatisfaction [12].

By extending policy-governed resilience to the Angular layer, the framework ensures that user-facing failure behaviors are as carefully managed and as easily updated as backend resilience policies. The microservice decomposition model described by Dragoni et al. [1] introduces a failure surface area at every service boundary; the proposed framework systematically addresses that surface area through a single, unified policy control plane rather than through per-service, per-layer configuration silos[13].

Limitations: The framework introduces a dependency on the Policy Engine as a control-plane component. While this caching allows enforcers to operate independently of Policy Engine availability, other factors around the governance and maturity of the Policy Engine affect performance [14]. For

instance, poorly designed policies may create excessive load as the Policy Engine retries requests too aggressively. Interfacing with legacy systems not based on Spring Boot and Angular may require the development of custom adapters, which should be taken into account when planning migration efforts.

Future Work: Providing real-time metric streams as input to the reinforcement learning agent proposing the policy mutations would allow us to reduce latency from API failure detection to policy mutation. Additional protocol support for GraphQL and WebSocket connections would allow applications using these protocols to use the framework as well [15]. Smoothly integrating with chaos engineering platforms enables automated validation of policy effectiveness and closes the loop between design-time resilience and operational assurance.

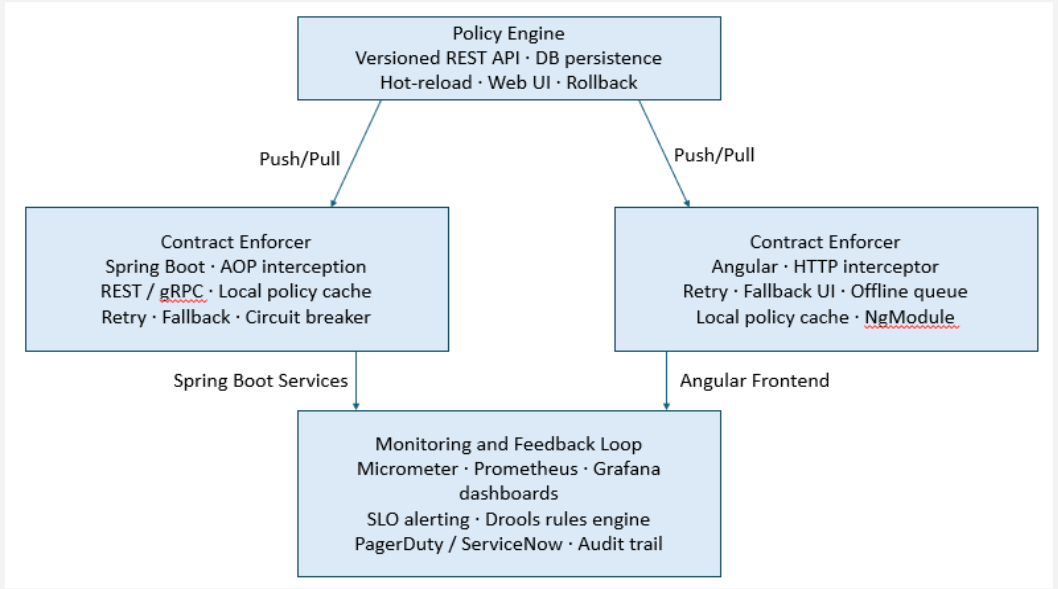


Figure 1: *Architecture of the Policy-Driven Adaptive Resilience Control Plane for Spring Boot Microservices and Angular Frontends [3, 5, 7-9]*

Table 1: *Qualitative Performance Comparison of Resilience Approaches Under Controlled Fault Injection Scenarios [3, 5]*

Approach	MTTR	Error Rate	Throughput	99th Percentile Latency	Policy Update Latency	CPU Overhead	Memory Overhead
Static Circuit Breakers	Highest among all evaluated configurations	Highest among all evaluated configurations	Lowest among all evaluated configurations	Highest among all evaluated configurations	Requires full service redeployment	Low	Low
Resilience4j	Moderately reduced compared to static approaches	Moderately reduced compared to static approaches	Marginally higher than static approaches	Moderately reduced compared to static approaches	Requires application restart	Moderate	Moderate
Istio/Envoy	Notably lower than code-embedded approaches	Lower than code-embedded approaches	Comparable to Resilience4j	Lower than code-embedded approaches	Supported via control plane config reload; no restart required	Higher than code-embedded approaches	Higher than code-embedded approaches

Table 2: *Feature-Level Comparison of Resilience Frameworks Across Adaptability, Coverage, and Observability Dimensions [2-6]*

Feature	Static Approaches	Resilience4j	Istio/Envoy	Proposed Framework
Runtime Policy Update	No	Partial	Partial	Yes
Contract-Oriented	No	No	No	Yes
Full-Stack (Backend + UI)	No	No	No	Yes
Monitoring Feedback Loop	No	No	Partial	Yes
Hot Reload	No	Partial	Yes	Yes
Business Policy Support	No	No	No	Yes
Extensibility	Low	Medium	Medium	High
Observability	Low	Medium	Medium	High

8. Conclusions

Enterprise applications built on distributed microservice architectures with Angular frontends have a recurring problem with resilience: Fault tolerance mechanisms performed statically in code are structurally unfit for this task. This article offers a solution to this fundamental limitation in the form of an adaptive resilience framework whose behavior is externally specified as versioned contracts, dynamically maintained in a central Policy Engine, and enforced using Spring AOP and Angular HTTP interceptors based on an integrated Prometheus, Grafana and Drools-based feedback loop monitoring approach. By extending policy-governed fault tolerance to both backend services and browser-resident frontend clients, the framework provides a solution to the shortcomings of the per-layer configuration silos of existing solutions. In addition, its contract-oriented approach to fault tolerance decouples it from business logic, propagates policy changes to the system in near real time, and supports extensibility into domain-specific operational toolchains, e.g., observability and incident analysis. Future features include optimizing API policies with reinforcement learning, support for more protocols like GraphQL and WebSocket, and integration with chaos engineering applications for automated resilience testing.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.
- **Use of AI Tools:** The author(s) declare that no generative AI or AI-assisted technologies were used in the writing process of this manuscript.

References

[1] Nicola Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," arXiv:1606.04036v4, 2017. [Online]. Available: https://arxiv.org/pdf/1606.04036

[2] Netflix, "Netflix/Hystrix," GitHub Repository. [Online]. Available: https://github.com/Netflix/Hystrix

[3] Resilience4j, "Resilience4j is a fault tolerance library for Java™," Official Documentation. [Online]. Available: https://resilience4j.readme.io/

[4] Kubernetes, "Configure Liveness, Readiness and Startup Probes," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

[5] Istio Authors, "Istio Service Mesh Documentation," Official Documentation. [Online]. Available: https://istio.io/latest/docs/

[6] Spring.io, "Spring Cloud Circuit Breaker," Spring Projects. [Online]. Available: https://spring.io/projects/spring-cloud-circuitbreaker

[7] Prometheus Authors, "Prometheus – Monitoring System and Time Series Database," Official Documentation. [Online]. Available: https://prometheus.io/

[8] Grafana Labs, "Grafana: The Open and Composable Observability Platform,". [Online]. Available: https://grafana.com/

[9] Bimal Jha, "Unlocking Business Logic: A Guide to Drools Rule Engine," Medium, 2024. [Online]. Available: https://medium.com/@bimalkumarjha91/unlocking-business-logic-a-guide-to-drools-rule-engine-1af098ffabb9

[10] Sam Newman, "Building Microservices: Designing Fine-Grained Systems," O'Reilly Media, 2015. [Online]. Available: https://books.google.co.in/books?id=jjl4BgAAQBAJ&printsec=frontcover&source=gbs_atb#v=onepage&q&f=false

[11] Quintero, F. A., "Fluid dynamics-based VFX design: Trade-offs between visual realism, computational cost, and production timelines," *Journal of Computational Analysis and Applications*, 31(4), 2722–2736, 2023.

[12] Belhassen, A., "Closed-loop system identification and control of a 3D-printed soft robot using MATLAB and vision-based feedback," *Sarcouncil Journal of Applied Sciences*, 4(11), 31–39, 2024.

[13] Surana, S., "The analytical review as a core management tool: Techniques for identifying variances, ensuring accounting accuracy, and informing strategy," *Journal of Information Systems Engineering and Management*, 8(3), 1–10, 2023.

[14] Darteh, F. K., "Performance-based budgeting and the role of reliable revenue reports," *Sarcouncil Journal of Economics and Business Management*, 2(12), 8–16, 2023.

[15] Chhibber, R., "Strategic leadership in partner sales networks for enterprise market expansion," *Journal of International Crisis and Risk Communication Research*, 4(3), 467–475, 2021.