

ML-Based Preemptive Restarts: Process-Level Orchestration as a Lightweight Alternative to Containers for Node.js

Felix Desyatirikov
ITMO University
St. Petersburg, Russia
felixdesyatirikov@gmail.com

Abstract—This paper addresses the problem of hosting a large number of lightweight Node.js applications on a single server in situations where full-scale container orchestration (Kubernetes, Docker Swarm) introduces excessive overhead. A process-level orchestration approach is proposed in which each application runs as a separate operating system process managed by a single application server implemented in Kotlin with Spring Boot. The paper describes the architecture of the Node Orchestrator system, including a REST API for deployment, a reverse-proxy mechanism, automatic restart of crashed processes, and an automated monitoring module: anomaly detection in process metrics based on the Isolation Forest algorithm for preemptive restarts, and adaptive cooldown management based on failure history. The experiments demonstrate that the prototype works correctly and consumes fewer resources than the container-based approach. The results suggest that process-level orchestration combined with automated monitoring components can serve as a promising lightweight alternative under the evaluated conditions for hosting small Express.js services.

Keywords—*process orchestration, Node.js, containerization, Docker, Kubernetes, Spring Boot, reverse proxy, PaaS, ALM, anomaly detection, ML, adaptive cooldown*

I. INTRODUCTION

The software industry has for some years now been moving steadily toward microservice architectures [1], in which an application is decomposed into a set of autonomous services that communicate over the network. The Node.js platform [2], with its non-blocking I/O model and single-threaded event loop, has become one of the most widely adopted server-side environments for implementing such services. According to the Stack Overflow Developer Survey 2024 (*Stack Overflow Developer Survey 2024*, <https://survey.stackoverflow.co/2024>), Node.js is used as a web technology by 42.73% of professional developers, making it the most popular server-side framework, while Express.js remains the de facto standard for building REST (REpresentational State Transfer) APIs (Application Programming Interface).

For managing the lifecycle of multiple services – deployment, scaling, monitoring, and recovery – container orchestration systems have become the de facto standard: Kubernetes [3], Docker Swarm [4], Nomad, and others [5]. Containerization provides reproducible environments, dependency isolation, and horizontal scaling. However, in scenarios where tens, hundreds, or thousands of small Express.js applications need to run on a single physical or virtual host (for instance, in educational or hosting platforms), the container approach introduces significant overhead: every container carries a full copy of the runtime

environment, consumes additional memory for filesystem layers, and requires a cluster-level orchestrator.

An alternative to container-based isolation is process-level orchestration – an approach in which each application runs as an ordinary operating system process, and a controlling application server takes on the responsibilities of deployment, traffic routing, and health monitoring. This approach is well known from PaaS (Platform-as-a-Service) platforms [6] such as Heroku [7] and Nodejitsu, yet it has received little formal treatment in the academic literature, in contrast to its adoption in production tooling, and direct comparisons with the container-based approach remain scarce.

The aim of this research is to design and experimentally evaluate a lightweight Node.js application orchestrator based on process-level isolation, and to compare it against the container-based approach.

To achieve this aim, the following objectives were set:

- To review existing approaches to Node.js application orchestration and identify their limitations.
- To design the architecture of an application server with a process-based isolation model.
- To implement a prototype of the Node Orchestrator system in Kotlin using Spring Boot.
- To develop an automated monitoring module based on the Isolation Forest algorithm for anomaly detection and adaptive restart control.
- To carry out an experimental evaluation of the prototype and assess its viability.
- To perform a comparative analysis of resource consumption between the process-based and container-based approaches.

As a result of this work the Node Orchestrator prototype has been designed and implemented – a Kotlin/Spring Boot server application that provides a full management cycle for Node.js applications on a single host, extended with an automated monitoring module. The experimental evaluation confirmed that the prototype works as intended and reduces RAM (Random Access Memory) consumption by a factor of 2.4 to 2.6 compared with the container-based approach.

The research contributes on two levels. At the conceptual level it offers a structured comparison of the design space for single-host hosting of lightweight Node.js applications, a compact implementation-independent formalization of the process-level orchestration model, a quantitative characterization of the resource gap between process-level and container-per-application deployment

together with an analysis of its source, a demonstration that a lightweight anomaly detection model can operate inside the orchestration control loop, and a parameterized generalization of truncated exponential backoff for restart control. At the practical level it delivers a working prototype with validated default configuration values, a technique for binding an application to an orchestrator-assigned endpoint without modifying its source code, a two-level routing scheme with measured latency and throughput cost, and an explicitly delimited applicability envelope accompanied by a hardening roadmap. The contributions are stated in full at the end of the Conclusion.

II. THEORETICAL PART

To ground the proposed solution in a clear conceptual framework, we first define the relevant domain and its fundamental terms.

A. Research Area and Key Concepts

This research sits at the intersection of two areas: Application Lifecycle Management (ALM) and Platform-as-a-Service (PaaS) [6]. By application orchestration we mean the automated management of deployment, startup, shutdown, traffic routing, and recovery from failures.

The key managed entity in this work is an instance of a Node.js application – a server process that executes Express.js code and serves HTTP requests on a dedicated port. The orchestrator manages many such instances on a single host.

It is worth noting that process-level orchestration is not limited to web service hosting. It can also be applied effectively in related scenarios where low overhead and fast process startup matter: serving machine learning models with low latency, running background jobs, building API proxy gateways, and deploying on edge and IoT (Internet of Things) devices where containerization is impractical due to limited resources.

B. Existing Solutions

Container orchestration. Kubernetes [3] is the dominant solution for managing containerized applications. Each service is packaged into a Docker image [4] and deployed as a Pod – the smallest schedulable unit. Kubernetes handles autoscaling, load balancing, self-healing, and declarative configuration. Lightweight distributions such as K3s [8] and MicroK8s target edge devices and small clusters.

PaaS platforms. Heroku [7] uses the dyno model – isolated containers each running a single application instance. Similar models are used by Railway, Render, and Fly.io. These platforms abstract away the infrastructure, but they are commercial cloud services and offer no control over the host itself.

Process managers. Tools such as PM2 and forever start Node.js applications as system processes and restart them on failure. However, they do not provide a REST API for dynamic deployment, do not handle traffic routing, and do not manage ports.

Application servers with process-level isolation. Between process managers and container platforms lies a class of tools that run applications as ordinary host processes while themselves terminating and routing HTTP traffic. NGINX Unit (NGINX Unit, <https://unit.nginx.org/>) is configured entirely through a RESTful JSON API, applies

changes at runtime without interruption, and dispatches requests by configurable matching rules; Phusion Passenger (Phusion Passenger, <https://www.phusionpassenger.com/>) spawns and supervises Node.js processes behind nginx or Apache. Both confirm the practical viability of the execution model adopted here. They differ in two respects: their APIs register references to code already present on the host, rather than accepting a deployable artifact, and restarts are triggered by process termination rather than by observed degradation.

System supervisors. Tools such as systemd [9], supervisord (*Supervisor: A Process Control System*, <https://supervisord.org/>), and runit (runsv) provide reliable startup, restart, and monitoring for arbitrary processes at the operating system level. They support declarative service definitions, dependency management between services, log redirection, and resource limits (in the case of systemd, via built-in cgroups integration). However, these tools are geared toward static configuration: adding a new service requires creating a unit file or configuration entry and reloading the supervisor. They do not provide a REST API for dynamic deployment from archives, do not route HTTP traffic by alias, and do not manage a port pool. That said, their process management model (start, monitor, restart with backoff) served as an architectural reference when designing Node Orchestrator.

Serverless platforms. AWS Lambda (Amazon Web Services, *AWS Lambda | Serverless Compute*, <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>) and Google Cloud Functions (*Google Cloud Functions Documentation*, <https://cloud.google.com/functions/docs>) implement the Function-as-a-Service (FaaS) model [10], in which each incoming request is handled by an isolated function instance that the cloud provider scales automatically. For Node.js applications this removes the need to manage infrastructure, but it comes with significant limitations: hard quotas on concurrent instances (1000 per region by default for AWS Lambda; 100 to 1000 for Google Cloud Functions depending on the generation), execution time limits (15 min for Lambda, 9 min for Cloud Functions 1st gen), and the cold start problem – spinning up a new Node.js environment when no warm instance is available adds 100 to 500 ms of latency to the first request. In addition, serverless platforms do not support long-lived WebSocket connections, which limits their applicability for real-time applications. At high call volumes, the cost can also significantly exceed that of process-based or container-based hosting. Table 1 compares existing solutions.

TABLE I. COMPARISON OF APPROACHES TO NODE.JS APPLICATION ORCHESTRATION

Criterion	Approach					
	K8s	PaaS	Serverless	App servers	PM2	systemd
Overhead	High	Med	Low	Low	Low	Low
REST API	Yes	Yes	Yes	Yes	No	No
Routing	Yes	Yes	Yes	Yes	No	No
Auto-recovery	Yes	Yes	Yes	Yes	Part.	Yes
Host control	Full	No	No	Full	Full	Full
Scale limits	No	Paid	1000/region	No	No	No
WebSocket	Yes	Yes	No	Yes	Yes	Yes

Dynamic deploy	Yes	Yes	Yes	Part.	No	No
Complexity	High	Low	Low	Med.	Low	Med.

As Table 1 shows, no single approach combines all of the following at once: low overhead, a REST API, built-in traffic routing, automatic recovery, full control over the host, no hard scaling quotas, support for persistent connections, and dynamic deployment of a self-contained code artifact. Serverless platforms offer an attractive autoscaling model. However, they cap the number of concurrent instances. They do not support long-lived connections. They also impose cold-start latency. Application servers with process-level isolation come closest. They satisfy every criterion except the last one. Table 1 records that criterion as partial. The reason is that their control APIs register references to code that must already be present on the host. One further difference is not captured in Table 1. None of the listed approaches derives restart decisions from observed process metrics. Recovery is triggered after a process has terminated, not upon detected degradation.

C. Proposed Solution

A process-level orchestration approach is proposed, which can be formalized as follows.

Let H be the host machine with a set of available ports:

$$P = \{p_{min}, \dots, p_{max}\}.$$

Let A be the set of deployed applications:

$$A = \{a_1, a_2, \dots, a_n\},$$

where n is the total number of deployed applications.

For each application a_i , an injective mapping function f

$$f: A \rightarrow P$$

assigns a unique port $f(a_i)$, to each application, and a state function s

$$s: A \rightarrow S$$

captures its current status, where $S = \{Running, Stopped, Restarting\}$ is the set of possible instance states.

1) PUBLISH(archive, alias) $\rightarrow$ unpack the archive, allocate a port $p=f(a)$, start the process, and register the route alias $\rightarrow p$.

2) STOP(id) $\rightarrow$ terminate the process, release the port, and remove the route.

3) STATUS(id) $\rightarrow$ return the current state $s(a_i)$.

4) PROXY(alias, request) $\rightarrow$ look up the port by alias and proxy the HTTP request to 127.0.0.1: p .

Automatic recovery is implemented by a background job with a fixed check interval and a restart cooldown that prevents rapid restart loops for broken instances:

$$T_{check}, T_{cooldown}.$$

The main advantage of this approach is that there is no intermediate containerization layer. All applications share a common Node.js installation and a common node_modules directory, which substantially reduces both disk and memory consumption.

D. Machine Learning Methods in Orchestration

The Existing orchestration systems, including Kubernetes and PM2, take a reactive approach to recovery: a failure is detected only after the process has already crashed, at which point the system restarts it with a fixed delay. This approach has two notable drawbacks: downtime between the crash and its detection (determined by the check interval), and a uniform restart strategy that ignores the individual failure history of each instance.

Proactive recovery itself has an established research background. Software rejuvenation was introduced by Huang et al. [11]. The idea is to terminate a long-running process gracefully and restart it in a clean state before a failure occurs. Garg et al. [12] put this on a measurement basis. They monitored operating system resources during execution and estimated the time to resource exhaustion statistically. The wider problem of predicting failures from runtime monitoring data is surveyed by Salfner et al. [13]. The monitoring module described below applies this line of work to process-level orchestration. Two things differ from the classical setting. The decision is taken per application instance rather than per host. The detector also runs inside the orchestrator's own health-check loop, so the component that observes degradation is the one that performs the restart.

In this work, process-level orchestration is extended with two automated monitoring components: an anomaly detection module and an adaptive cooldown module.

The anomaly detection module is based on Isolation Forest [14] – an ensemble algorithm that identifies anomalous observations by constructing random trees. For each instance a_i , a feature vector is formed:

$$x_i(t) = (cpu_i, rss_i, latency_i, err_{rate_i})$$

where cpu is the process CPU usage (%); rss is the resident memory consumption (MB); $latency$ is the mean response time over the last interval (ms); err_rate is the fraction of HTTP responses with 5xx status codes. The model is trained on metrics collected during normal operation and computes an anomaly score [14]:

$$score(x) = 2^{-\frac{E[h(x)]}{c(m)}},$$

where $E[h(x)]$ is the mean isolation depth of observation x across all trees in the ensemble; $c(m)$ is a normalizing coefficient that depends on the training sample size m . Score values close to 1 indicate an anomaly. When the score exceeds a threshold θ , the orchestrator triggers a preemptive restart of the instance before the process actually crashes.

The adaptive cooldown module replaces the fixed restart delay with a dynamically computed value that takes into account the failure history of the specific instance. The adaptive cooldown generalizes the truncated exponential backoff scheme [15], as implemented in the Kubernetes CrashLoopBackOff mechanism [16], by making the growth factor α and the reset window W configurable. The cooldown for instance a_i after its k -th failure is given by the following exponential backoff formula:

$$T_{cooldown}^{(k)} = T_{base} \min(\alpha^{k-1}, T_{max}/T_{base}),$$

where T_{base} is the base delay (default: 5 s); α is the growth factor (default: 1.5); T_{max} is the maximum delay (default: 300 s); k is the number of failures within a sliding time window W (default: 1 hour). The counter k is reset if the instance runs continuously for longer than W without failures. This way, consistently stable instances are restarted with minimal delay, while chronically broken ones face an ever-increasing delay, which avoids wasting resources on pointless restart loops.

III. EXPERIMENTAL PART

This section outlines the principal tools and technology stack chosen for implementing and deploying that prototype.

A. Tools and Technology Stack

The following technology stack was chosen for the prototype.

- Kotlin – a statically typed language for the JVM (Java Virtual Machine) that offers concise and safe code.
- Spring Boot 3 – a framework for building production-ready applications on the JVM with minimal configuration.
- H2 Database – an embedded relational DBMS (Database Management System) operating in file mode.
- Nginx – a high-performance HTTP server and reverse proxy.
- Node.js 20 / Express.js – the target runtime for hosted applications.
- Docker / Docker Compose – used to package the orchestrator itself.

Kotlin and Spring Boot were chosen for their mature ecosystem, the database abstraction layer provided by Spring Data JDBC, the Spring MVC stack for REST APIs, and the scheduling mechanism (@Scheduled) for background health checks of instances.

B. System Architecture

The Node Orchestrator architecture is organized as a three-tier model (Fig. 1).

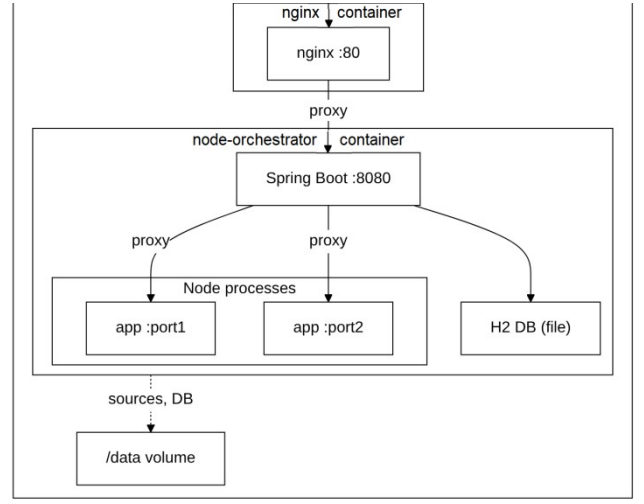


Fig. 1. Deployment architecture of the Node Orchestrator system.

The outer tier consists of nginx, which accepts HTTP requests on port 80 and routes them either to the orchestrator's REST API (prefix /api/v1/) or to a specific application by its alias. The middle tier is the Spring Boot application server on port 8080, which implements the orchestration business logic. The bottom tier is the set of Node.js processes, each bound to a unique port within the configured range.

C. Instance Lifecycle

Each application instance transitions through the states, shown in Fig. 2.

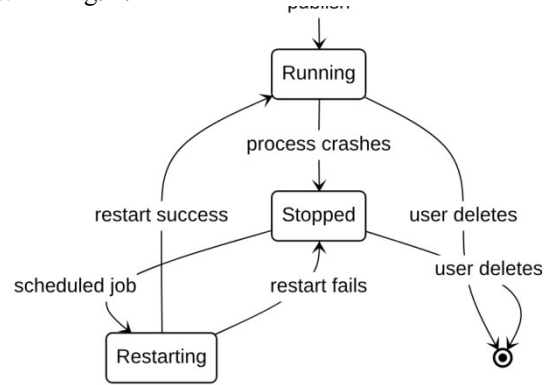


Fig. 2. State diagram of an application instance.

The Running state means that the Node.js process is up and serving HTTP requests. When the process terminates abnormally, the state transitions to Stopped. A background job with a configurable interval checks the state of all instances; when a stopped instance is found, the orchestrator attempts to restart it (the Restarting state), honoring the cooldown delay to avoid restart loops for broken applications.

D. REST API of the Application Server

The server exposes a REST API; its endpoints are listed in Table 2.

TABLE II. REST API ENDPOINTS

Method	Path	Description
POST	/api/v1/publish	Deploy an application from a tar.gz archive
DELETE	/api/v1/stop/{id}	Stop and remove an instance
GET	/api/v1/status/{id}	Get the status of an instance
GET	/api/v1/list	Paginated list of instances

Deployment is performed via a multipart/form-data request containing a tar.gz archive with the application source code, a unique alias, and the path to the entry file (index.js by default). The server unpacks the archive, allocates a free port from the pool, spawns a Node.js process, and registers the route in the alias cache.

E. Traffic Proxying Mechanism

User traffic is routed to applications via a two-level scheme. nginx (the first level) accepts requests and decides on the routing strategy:

- by subdomain: a request to `http://{alias}.domain.com/` is forwarded by nginx to Spring Boot, which extracts the alias directly from the request URL;
- by path: a request to `http://domain.com/{alias}/path` is forwarded in the same manner, with the alias extracted from the first URI segment.

Spring Boot (the second level) looks up the port in a thread-safe cache – `ConcurrentHashMap<alias, port>` – and proxies the request to `127.0.0.1:{port}` using Spring RestClient. The two-level scheme separates responsibilities: nginx handles TLS termination and static content, while Spring Boot takes care of the routing business logic.

F. Port Management and UNIX Sockets

The port range is defined in the configuration (runner.port_ranges) and can include several ranges with exclusions. The implementation uses a port pool: a port is taken from the pool on deployment and returned on shutdown. This guarantees that no two instances collide and allows the system to adapt to host-level constraints (for example, when some ports are already occupied by other services).

Binding an instance to its assigned port requires no changes to the Node.js application source code. The orchestrator uses Node.js `--require` to inject a middleware that intercepts calls to `net.Server.prototype.listen` and replaces the argument with the target address taken from the OUT environment variable (an arbitrary identifier chosen for the injected listener target, not an acronym).

Port binding middleware (out-inject.js):

```
const net = require('net');
const originalListen = net.Server.prototype.listen;
const forcedOut = process.env.OUT;

net.Server.prototype.listen = function (...args) {
  const firstArg = args[0];
  if (typeof firstArg === 'object' && firstArg !== null) {
    return originalListen.call(this,
      { ...firstArg, port: forcedOut }, ...args.slice(1));
  }
  return originalListen.apply(this,
    [forcedOut, ...args.slice(1)]);
}
```

```
};
```

This approach transparently redirects any Express.js application to the port assigned by the orchestrator. The same mechanism may be implemented for binding to a UNIX domain socket. In some cases when scaling up to thousands of instances, allocating TCP ports may become inconvenient (there is a limit of roughly 64K ports and a need for alias → port mapping).

The OUT variable may contain a path to a UNIX socket (for example, `/tmp/node-orch/myapp.sock`), which eliminates the need for a port pool, reduces latency by avoiding the TCP stack, and simplifies the nginx configuration (proxy_pass unix:... directive). However, in case of that big amount of instances the more common issue will be with the compute capacity of the physical node. Therefore, the UNIX domain socket mechanism is not used by default for better maintainability and observability of the presented solution.

G. Failure Recovery

The system implements two recovery mechanisms. The first is proactive monitoring: a background job (@Scheduled) with a configurable interval (5 s by default) checks whether each instance's process is alive and triggers a restart when necessary. The second is startup recovery: when the orchestrator itself starts, it checks the state of all previously registered instances in the H2 database; if a process is unresponsive, the instance is recreated with a newly assigned port. On a graceful stop (DELETE /stop), the process receives SIGTERM for clean shutdown and resource release; if it does not terminate within the grace period (10 s by default), SIGKILL is sent.

H. Implementation of the Anomaly Detection Module

The anomaly detection module is implemented as the AnomalyDetectorService component, integrated into the background monitoring loop. On each check interval the service collects metrics from every running instance: CPU usage, Resident Set Size (RSS), average latency of recent requests, and the fraction of 5xx responses. Metrics are obtained from Node.js processes through `/proc/{pid}/stat` for CPU and RSS, while latency and response codes are captured via a built-in middleware that the orchestrator injects when an instance starts.

For each instance the module keeps a sliding window of metrics with size of 100 observations. The Isolation Forest model is trained on the first 100 observations (the warm-up period) and retrained every 500 observations to adapt to changes in the normal load profile [17]. The implementation uses the Smile library [18] (Statistical Machine Intelligence and Learning Engine) for the JVM, which provides an efficient Isolation Forest implementation.

When an anomaly is detected (score > θ , where $\theta = 0.65$ by default), the module emits an AnomalyDetectedEvent. If the anomalous state persists across three consecutive checks, the orchestrator performs a preemptive restart of the instance. Requiring three consecutive triggers reduces false positives caused by short-lived load spikes.

I. Adaptive Cooldown Implementation

The adaptive cooldown module is implemented in the CooldownStrategy class, which replaces the instance.restart.cooldown-ms parameter. For each instance the module maintains a failure log – a list of crash timestamps within a sliding window $W = 3600$ s. On each

failure, the number of recent failures k within the window is computed, and the restart delay is determined by the exponential growth formula.

Module configuration is exposed via application.yaml:

```
instance:
  cooldown:
    base-ms: 5000      # T_base - base delay
    multiplier: 1.5    #  $\alpha$  - growth factor
    max-ms: 300000     # T_max - delay cap (5 min)
    window-ms: 3600000 # W - sliding reset window (1 hour)
```

Thus, an instance's first failure results in a 5 s delay, the second in 7.5 s, the third in 11.25 s, and so on up to the maximum of 300 s. If the instance runs without failures for longer than W , the counter resets, and the next failure will again incur T_{base} . This strategy allows stable instances to be recovered quickly after isolated failures while avoiding wasted resources on restart loops for broken applications.

J. Code-level Implementation

The project is implemented in Kotlin using the Gradle build system. Key implementation fragments are shown below.

Dependency configuration (build.gradle.kts):

```
plugins {
    kotlin("jvm") version "2.1.0"
    kotlin("plugin.spring") version "2.1.0"
    id("org.springframework.boot") version "3.4.3"
}
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-webmvc")
    implementation("org.springframework.boot:spring-boot-starter-data-jdbc")
    implementation("org.apache.commons:commons-compress:1.27.0")
    runtimeOnly("com.h2database:h2")
}
```

Nginx configuration (nginx.conf):

```
upstream backend_spring_app {
    server node-orchestrator:8080;
}

map $host $subdomain_alias {
    default "";
    ~^(?<sub>[^\.]+\.)\.localhost$ $sub;
}

server {
    listen 80;
    server_name localhost ~^(?<subdomain>.+)\.localhost$;

    # Orchestrator API - direct pass-through
    location /api/ {
        proxy_pass http://backend_spring_app;
    }

    # Subdomain routing: alias.localhost → /api/v1/alias/{alias}
    location / {
        if ($subdomain_alias != "") {
            rewrite ^(.*)$ /api/v1/alias/$subdomain_alias$1 break;
            proxy_pass http://backend_spring_app;
        }
        try_files $uri @path_alias;
    }

    # Path routing: localhost/{alias}/... → /api/v1/alias/{alias}/...
    location @path_alias {
        rewrite ^/(?<path_alias>[^\.]+)(?<remaining>/?.*)$ /api/v1/alias/$path_alias$remaining break;
        proxy_pass http://backend_spring_app;
    }
}
```

The configuration implements two routing modes. The map directive extracts the subdomain from the Host header (for example, myapp from myapp.localhost) and stores it in the \$subdomain_alias variable. If the subdomain is present, the request is rewritten to /api/v1/alias/{alias} and proxied to Spring Boot. If no subdomain is present, the @path_alias fallback is used, which extracts the alias from the first URI segment (for example, /myapp/endpoint → /api/v1/alias/myapp/endpoint). In both cases, Spring Boot

receives a uniform request of the form /api/v1/alias/{alias}/..., which simplifies the internal routing logic.

IV. EVALUATION OF RESULTS

This section describes the test environment, measurement methodology, and the set of scenarios used to benchmark the prototype against the container-based alternative.

A. Experimental Methodology

The prototype was evaluated on a virtual machine with the following specifications: Ubuntu 24.04, 4 vCPUs (Intel Xeon E5-2686 v4), 8 GB of RAM, and a 40 GB SSD. The machine had Docker 27.x, Node.js 20.x, and Java 21 (Eclipse Temurin) installed. A test Express.js application returning a JSON payload with environment information was prepared for testing.

RAM consumption was measured with the smem utility, which reports actual RSS usage per process, excluding shared pages. For the container-based approach we additionally used docker stats --no-stream. All measurements were taken after the instances had stabilized (30 s after the last instance started) while idle (no active load), which reflects the baseline overhead of each approach. Each scenario was repeated 5 times; the tables report mean values with standard deviations.

Proxy latency was measured with the wrk tool (2 threads, 50 connections, 30 s) against the test application via nginx. Recovery time after a crash was measured as the interval between sending kill -9 to the target process and receiving a successful HTTP response from the recreated instance.

Test scenarios:

- 1) Deploying an application via POST /api/v1/publish.
- 2) Checking the status via GET /api/v1/status/{id}.
- 3) Routing an HTTP request to an application by alias.
- 4) Forcefully killing the process (kill -9) and observing automatic recovery.
- 5) Simultaneous deployment of 10, 20, and 50 instances – measuring RAM and time.
- 6) Load testing of proxy latency (wrk, 50 connections, 30 s).
- 7) Evaluating the anomaly detection module – simulating memory leaks and latency spikes.
- 8) Evaluating the adaptive cooldown against a fixed delay and a Kubernetes-style exponential backoff – 10 consecutive crashes of an instance.

B. Experimental Results

All baseline scenarios (1–4) completed successfully. Deploying a single application takes between 1.2 and 2.8 s (mean 1.9 s, standard deviation $\sigma = 0.5$ s), including archive extraction, port allocation, and process startup. Automatic recovery after kill -9 fires within the configured check interval: the mean recovery time was 6.3 s ($\sigma = 1.1$ s) with a 5-second check interval.

The RAM consumption measurements (scenario 5) are reported in Table 3. Values are means over 5 runs $\pm$ standard deviation.

TABLE III. RAM CONSUMPTION (RSS) FOR VARYING NUMBERS OF INSTANCES

Instances	Process RAM, MB	Container RAM, MB	Ratio	Deploy time, s
10	342 ± 18	814 ± 35	2.38×	11.4 ± 1.2
20	638 ± 27	1523 ± 62	2.39×	24.7 ± 2.1
50	1384 ± 54	3612 ± 108	2.61×	67.2 ± 4.8

The container-based numbers were obtained by deploying the same application in parallel across separate Docker containers (`docker run --memory=512m`) on the same machine. Container RAM usage was measured via `docker stats --no-stream --format '{{.MemUsage}}'` and summed over all containers. As Table 3 shows, the process-based approach consumes 2.4 to 2.6 times less RAM, thanks to the shared Node.js runtime and common `node_modules` directory.

Proxy latency measurements (scenario 6) are shown in Table 4. Two-level proxying adds an average of 1.62 ms to request latency, which is an acceptable overhead for most web applications. The 19.3% drop in throughput is due to the extra network hop through Spring Boot; for applications with low request rates (a typical scenario for hosting educational and demo services), this overhead is negligible.

TABLE IV. PROXY LATENCY AND THROUGHPUT

Metric	Direct access	Via proxy (nginx + Spring Boot)	Overhead
Mean latency, ms	1.12 ± 0.08	2.74 ± 0.15	+1.62
P99 latency, ms	3.41 ± 0.22	6.18 ± 0.38	+2.77
Requests/s (RPS)	18,420 ± 340	14,870 ± 280	-19.3%

Two-level proxying adds an average of 1.62 ms to request latency, which is an acceptable overhead for most web applications. The 19.3% drop in throughput is due to the extra network hop through Spring Boot; for applications with low request rates (a typical scenario for hosting educational and demo services), this overhead is negligible.

C. Evaluation of the Anomaly Detection Module

For evaluating the anomaly detection module (scenario 7), 20 instances were prepared running under regular load (wrk, 10 connections) for 10 minutes to train the model. Anomalies were then artificially induced in 5 of the instances: two exhibited linear RSS growth (simulating a memory leak, +5 MB/s), two showed increased response latency (setTimeout injections ranging from 100 to 2000 ms), and one produced a rising fraction of 5xx responses (random errors with probability growing from 0 to 80%). The remaining 15 instances continued normal operation. Classification results are shown in Table 5.

TABLE V. ANOMALY DETECTION RESULTS ($\theta = 0.65$, 3-IN-A-ROW RULE)

Metric	Value
True Positive (anomaly correctly detected)	5 of 5
False Positive (false alarm)	1 of 15
Mean time to detection (from anomaly onset)	18.4 ± 3.2 s
Precision	0.833

Recall	1.000
F1-score	0.909

All 5 anomalous instances were detected and preemptively restarted before the process actually crashed. One false positive occurred on an instance with a brief CPU spike caused by V8 garbage collection – the three-consecutive-triggers rule did not filter this case out because the garbage collection (GC) intervals happened to line up with the check period. The mean detection time of 18.4 s is substantially shorter than the typical time from the onset of degradation to process failure (60–180 s for memory leaks), which supports the practical value of the preemptive approach.

D. Evaluation of the Adaptive Cooldown

To evaluate the adaptive cooldown (scenario 8), we prepared an instance with intentionally broken code (throwing an exception after a random interval of 2–10 s from startup). The instance was subjected to 10 consecutive crashes. Actual restart delays and total downtime were measured. For comparison, the same scenario was run under two baseline strategies. Both are driven by the same cooldown implementation. The first is a naive fixed cooldown of 30 s. The second is a Kubernetes-style truncated exponential backoff. Its initial delay is 10 s, it doubles on each failure, and it is capped at 300 s. This reproduces the actual restart behaviour of the Kubernetes `CrashLoopBackOff` mechanism [16]. Results are presented in Table 6.

TABLE VI. COMPARISON OF COOLDOWN STRATEGIES (10 CONSECUTIVE CRASHES)

Metric	Fixed (30 s)	Kubernetes exp (10...300 s)	Adaptive (5...300 s)
1st restart delay, s	30	10	5
5th restart delay, s	30	160	25
10th restart delay, s	30	300	192
Total downtime, s	300	1810	567
Total restart attempts	10	10	10
CPU spent on restarts, %	8.2 ± 0.4	5.1 ± 0.3	4.1 ± 0.3

Table 6 compares three restart strategies driven by the same cooldown implementation: a naive fixed delay, a Kubernetes-style exponential backoff, and the proposed adaptive strategy. Total downtime is deliberately not the primary optimization target here: for a permanently broken instance a longer accumulated delay is the intended behaviour, since repeatedly restarting hopeless code only wastes resources. This is reflected in CPU usage – both exponential strategies throttle restarts and spend markedly less CPU than the fixed strategy (4.1% for the adaptive strategy vs. 8.2% for the fixed one). The decisive metric is recovery latency in the common case of a stable instance suffering an isolated crash. Here the adaptive strategy recovers fastest: after 5 s, versus 10 s for the Kubernetes-style backoff and 30 s for the fixed delay. Moreover, unlike the fixed 10-minute reset window built into Kubernetes, the adaptive module exposes a configurable decay window W and growth factor α , letting operators tune how quickly the

failure counter forgets isolated crashes and how aggressively the delay escalates for chronically failing instances.

E. Discussion of Limitations

The process-based model has a number of limitations that need to be acknowledged. First, process-level isolation is weaker than container isolation [19]: applications share the host's filesystem and network stack. This raises security concerns that are especially serious in hosting scenarios: potential access to other instances' files, possible inter-process communication, and the chance that one process exhausts resources at the expense of the others. The current implementation provides basic isolation: each instance lives in its own directory with 0700 permissions (owner only), and the process is launched via ProcessBuilder with the option to run under a dedicated system UID. Stronger isolation can be added via namespace isolation through unshare (mount, pid, network namespaces), resource limits via cgroups v2, and AppArmor/seccomp profiles to restrict system calls. Without these additional measures, the system is only suitable for trusted environments (internal corporate platforms, educational setups) and not for public hosting of arbitrary user code.

Second, all instances share the same Node.js version and npm dependencies, which limits support for applications with conflicting dependencies. This is a deliberate architectural trade-off: a shared node_modules directory saves disk space (on the order of 5–10 GB for 50 instances compared with per-instance copies). It is worth noting that each Node.js process loads modules into its own address space regardless of where the files live on disk, so a shared node_modules does not directly reduce RAM consumption. The observed memory savings (2.4–2.6× compared with containers) come primarily from avoiding the overhead of the Docker daemon, overlay2 filesystem layers, and auxiliary container processes (containerd-shim, init).

Third, the orchestrator's architecture is not tightly coupled to Node.js. Process management, the port pool, proxying, and the adaptive cooldown all work with any runtime that spawns an HTTP server. Node.js specifics appear at only two points: the port-injection middleware (--require) and metrics collection through V8-specific mechanisms. Adapting the system to Python, Go, or Ruby would require replacing these components, but the core of the orchestrator would remain unchanged.

Fourth, scaling horizontally beyond a single host requires additional infrastructure (a load balancer, shared storage). These limitations define the scope in which the solution is most useful: hosting a large number of similar lightweight Express.js applications in a trusted environment on a single server.

V. CONCLUSION

This paper has examined the problem of efficiently hosting a large number of small Node.js applications on a single server. Existing container-orchestration solutions impose excessive overhead for this class of workloads. Serverless platforms (AWS Lambda, Google Cloud Functions) cap the number of concurrent instances and do not support persistent connections. Process managers (PM2) and system supervisors (systemd) provide neither dynamic deployment nor alias-based routing nor port management. Application servers with process-level isolation cover most

of these requirements, but their control APIs register references to code already present on the host rather than accepting a deployable artifact, and their restart decisions are not driven by observed process metrics.

A process-level orchestration approach has been proposed and implemented, embodied in the Node Orchestrator system – a Kotlin/Spring Boot server application. It covers the full lifecycle of Node.js applications on a single host: dynamic deployment from tar.gz archives via a REST API, traffic routing by subdomain and by path through a two-level proxy scheme (nginx plus Spring Boot), a port pool with support for multiple ranges and exclusions, automatic recovery of crashed instances, Isolation Forest-based anomaly detection over process metrics, and adaptive cooldown management based on each instance's failure history.

The experimental evaluation confirmed that the prototype works as intended and showed a 2.4× to 2.6× reduction in RAM usage compared with the container-based deployment of the same applications. The reduction is attributable to the absence of a container runtime layer rather than to the shared node_modules directory. The anomaly detection module makes it possible to catch process degradation (memory leaks, latency growth) before the process actually crashes, while the adaptive cooldown reduces downtime for stable instances after isolated failures and prevents resource-hungry restart loops for broken ones.

Thus, the main contributions of this study to computer science research and practice are as follows.

1. *A structured comparison of the design space.* The target workload is a large number of lightweight Node.js applications on one host. Six classes of tools are assessed: container orchestrators, PaaS platforms, serverless runtimes, application servers with process-level isolation, process managers, and system supervisors. A common set of operational criteria is applied. The criteria are overhead, availability of a REST API, built-in routing, automatic recovery, control over the host, scaling quotas, support for persistent connections, dynamic deployment, and operational complexity. The comparison makes explicit the combination of properties that no class provides jointly.

2. *A compact formalization of the process-level orchestration model.* Application instances are defined as a set. Port allocation is defined as an injective mapping. Instance states and the lifecycle operations PUBLISH, STOP, STATUS, and PROXY are expressed in the same notation. The notation is implementation-independent. It can be reused to describe or compare other process-level orchestrators.

3. *A quantitative characterization of the resource gap.* Process-level deployment is compared against container-per-application deployment on a single host. The measurement procedure is reported alongside the numbers. The source of the observed difference is analysed explicitly. It is attributed to the overhead of the container runtime, not to the shared node_modules directory. This clarifies under what conditions the effect should be expected to persist.

4. *A demonstration that lightweight machine learning fits into the orchestration control loop.* An Isolation Forest model is built over a four-dimensional metric vector: CPU,

RSS, latency, and error rate. The model runs inside the background health-check cycle. Under the fault-injection scenarios considered, it identifies degrading instances before they crash. This leaves time for a preemptive restart. Detection quality is reported for the selected threshold and confirmation rule.

5. *A parameterized generalization of truncated exponential backoff.* The growth factor and the failure-counter reset window become explicit configuration parameters. In the usual scheme they are fixed constants. The resulting behaviour is compared against a fixed delay and against a Kubernetes-style backoff. All three strategies face an identical crash sequence.

6. *A working prototype, Node Orchestrator.* The system described above is offered as a reference architecture. Default configuration values validated in the experiments are reported: health-check interval, anomaly threshold and confirmation rule, base delay, growth factor, delay cap, and reset window. They can be adopted as starting points.

7. *A technique for binding an application to an orchestrator-assigned endpoint.* The application source code is not modified. A middleware is injected through `--require`. It intercepts `net.Server.prototype.listen`. The same mechanism offers a migration path to UNIX domain sockets. This matters where TCP port allocation becomes inconvenient.

8. *A two-level routing scheme.* nginx forms the outer level and the application server the inner one. Both subdomain-based and path-based addressing are supported. The latency and throughput cost is measured. Practitioners can therefore judge whether the separation of concerns is worth the extra hop.

9. *An explicitly delimited applicability envelope.* The approach suits a trusted environment, a homogeneous set of lightweight applications, and a single host. These conditions are stated directly. A hardening roadmap is given for going beyond them: namespace isolation via unshare, resource limits via cgroups v2, AppArmor and seccomp profiles, and execution under a dedicated UID.

Future work includes: support for per-application package.json files with individual dependency installation; cgroups-based resource limits for process isolation; integration with external monitoring systems (Prometheus, Grafana); horizontal scaling across multiple hosts with a centralized load balancer; and extending the anomaly detection module toward predictive autoscaling based on time-series load forecasting.

ACKNOWLEDGMENTS

LLMs were used solely for language polishing and stylistic improvements in the preparation of this manuscript. All technical content, ideas, analyses, and substantive writing are the original work of the author.

REFERENCES

- [1] Newman S. *Building Microservices*, O'Reilly Media, Inc., 2021, 612 p.
- [2] Tilkov S., Vinoski S. Node.js: Using JavaScript to Build High-Performance Network Programs, *IEEE Internet Computing*, 2010, vol. 14, no. 6, pp. 80–83. DOI: 10.1109/MIC.2010.145.
- [3] Burns B., Grant B., Oppenheimer D., Stenberg E., Wilkes J. Borg, Omega, and Kubernetes, *ACM Queue*, 2016, vol. 14, no. 1, pp. 70–93. DOI: 10.1145/2890784.
- [4] Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment, *Linux Journal*, 2014, no. 239, article 2.
- [5] Casalicchio E. Container Orchestration: A Survey Systems Modeling, *Methodologies and Tools* / Eds by A. Puliafito, K. Trivedi, EAI/Springer Innovations in Communication and Computing, Springer, Cham, 2019, pp. 221–235. DOI: 10.1007/978-3-319-92378-9_14.
- [6] Pahl C. Containerization and the PaaS Cloud, *IEEE Cloud Computing*, 2015, vol. 2, no. 3, pp. 24–31. DOI: 10.1109/MCC.2015.51.
- [7] Middleton N., Schneeman R. *Heroku: Up and Running*, O'Reilly Media, Inc., 2013, 130 p.
- [8] Usman M., Ferlin S., Brunstrom A. Performance Analysis of Lightweight Container Orchestration Platforms for Edge-Based IoT Applications, *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*, Rome, Italy, 04–07 December 2024, 2024, pp. 321–332, DOI: 10.1109/SEC62691.2024.00032.
- [9] Sievers K., Poettering L. Beyond Init: systemd, *Proceedings of Linux Plumbers Conference*, November 2010. URL: <https://github.com/systemd/systemd>.
- [10] Baldini I., Castro P., Chang K. Serverless Computing: Current Trends and Open Problems, *Research Advances in Cloud Computing* / Eds by S. Chaudhary, G.Somani, R.Buyya, Springer Singapore, 2017, pp. 1–20. DOI: 10.1007/978-981-10-5026-8_1.
- [11] Huang Y., Kintala C., Kolettis N., Fulton N. D. Software Rejuvenation: Analysis, Module and Applications, *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS-25)*, Pasadena, CA, USA, 27–30 June 1995, 1995, pp. 381–390. DOI: 10.1109/FTCS.1995.466961.
- [12] Garg S., van Moorsel A., Vaidyanathan K., Trivedi K. S. A Methodology for Detection and Estimation of Software Aging, *Proceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE)*, Paderborn, Germany, 04–07 November 1998, 1998, pp. 283–292. DOI: 10.1109/ISSRE.1998.730892.
- [13] Salfner F., Lenk M., Malek M. A Survey of Online Failure Prediction Methods, *ACM Computing Surveys*, 2010, vol. 42, no. 3, article 10, pp. 1–42. DOI: 10.1145/1670679.1670680.
- [14] Liu F. T., Ting K. M., Zhou Z.-H. Isolation Forest, *Proceedings of the Eighth IEEE International Conference on Data Mining*, Pisa, Italy, 15–19 December 2008, 2008, pp. 413–422. DOI: 10.1109/ICDM.2008.17.
- [15] Metcalfe R. M., Boggs D. R. Ethernet: Distributed Packet Switching for Local Computer Networks, *Communications of the ACM*, 1976, vol. 19, no. 7, pp. 395–404. DOI: 10.1145/360248.360253.
- [16] The Kubernetes Authors. Pod Lifecycle – Container restart policy [Electronic resource], Kubernetes Documentation. URL: <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/>
- [17] Zhang W., Yang Q., Geng Y. A Survey of Anomaly Detection Methods in Networks, *Proceedings of the 2009 International Symposium on Computer Network and Multimedia Technology*, Wuhan, China, 18–20 January 2009, 2009, pp. 1–3. DOI: 10.1109/CNMT.2009.5374676.
- [18] Li H. Smile: Statistical Machine Intelligence and Learning Engine [Electronic resource]. URL: <https://haifengl.github.io>.
- [19] Kerrisk M. *The Linux Programming Interface*, No Starch Press, 2010, 1552 p.