

Cloud-Native Microservices Architecture: Designing Scalable Enterprise Applications Using .NET 8 and Docker Containers

Rambabu Nalluri

nram.cse@gmail.com

10200 independence pkwy apt 1712 Plano Tx 75025. ph: +1 9403129386

Abstract: Enterprise software increasingly has to absorb abrupt, order-of-magnitude changes in demand while releasing new features continuously. Monolithic applications, in which every capability is packaged, deployed and scaled as a single unit, struggle with both requirements. This paper presents a cloud-native microservices reference architecture for scalable enterprise e-commerce applications built with .NET 8, ASP.NET Core, gRPC and Docker containers orchestrated by Kubernetes. The platform is decomposed into four bounded-context services (Catalog, Shopping Cart, Discount and Ordering) that are exposed through a YARP API gateway and integrated through an event backbone, with polyglot persistence, CQRS, saga-based consistency and Open Telemetry-based observability. A systematic design methodology covering domain decomposition, container image optimisation, event-driven integration, resilience policies and autoscaling is defined and formalised with analytical models based on Little's Law, the Universal Scalability Law and composite availability. The reference implementation was evaluated against a functionally equivalent monolith using a load profile modelled on a Black Friday surge of ten times typical demand. At 5,000 requests per second the microservices deployment sustained a 95th-percentile latency of 196 ms with a 0.21% error rate, whereas the monolith degraded to 842 ms and 4.10% errors. Maximum SLO-compliant throughput rose from 2,850 to 6,420 requests per second, node-hours fell by 38.1%, and fault-injection experiments showed a measured availability of 99.96%. Native AOT compilation and chiseled base images reduced container size to 31 MB and cold-start time to 0.21 s for latency-critical services. The results indicate that .NET 8 and containers provide a practical and efficient foundation for cloud-native enterprise systems.

Keywords: *Cloud-Native Architecture, Microservices, .NET 8, Docker, Kubernetes, Event-Driven Architecture, Autoscaling, Scalability, Resilience, Enterprise Applications*

I. INTRODUCTION

Digital commerce has turned application demand into one of the least predictable quantities in enterprise computing. A retail platform that serves a few hundred requests per second on an ordinary day may be asked to serve ten times that volume during a seasonal promotion, a product launch or a flash sale. Fig. 3 illustrates this behaviour with request counts for a typical Friday compared with Black Friday: after a quiet early-morning period, traffic on the promotion day climbs steeply from 08h and remains roughly ten times higher than the typical-day level for the rest of the day. Infrastructure that is provisioned for the peak sits idle most of the year, while infrastructure provisioned for the average fails exactly when the business depends on it most. Designing applications that can follow such demand elastically, at acceptable cost and without sacrificing reliability, is therefore a central problem of enterprise architecture [20].

Traditional enterprise applications are usually built as monoliths: one code base, one deployable artefact and one shared relational database. This style is simple to develop and test in the early life of a product, and it remains an appropriate choice for small systems. As the system grows, however, its weaknesses become pronounced. Components cannot be scaled independently, so a hot path such as product search forces the whole application to be replicated together with its memory-hungry reporting and administration modules. A defect or memory leak in any module can bring down the entire process. Teams are coupled through a shared release train, long build cycles and a common technology stack, which slows the delivery of change [12], [13], [16].

The microservices architectural style addresses these problems by structuring an application as a suite of small, independently deployable services that are organised around business capabilities, own their data and

communicate through lightweight protocols [1], [2]. Combined with containers, declarative orchestration, continuous delivery and observability, the style forms the basis of what the Cloud Native Computing Foundation describes as cloud-native computing: scalable applications that run in dynamic environments and rely on loosely coupled systems that are resilient, manageable and observable [20]. The approach is not free of cost. Distributed systems introduce network latency, partial failure, eventual consistency and substantial operational complexity, and a poorly decomposed system can degenerate into a "distributed monolith" that combines the drawbacks of both styles [4], [18]. Concrete, quantitative guidance on how to design and operate such systems is still scarce for specific technology stacks.

The .NET platform has changed considerably in recent releases. .NET 8, a long-term-support release, offers a high-performance Kestrel web server, minimal APIs, first-class gRPC support, a container-aware garbage collector, built-in container publishing, non-root "chiseled" base images, Native ahead-of-time (AOT) compilation for ASP.NET Core workloads and native OpenTelemetry integration. Docker provides the packaging format that turns these services into immutable, portable images, and Kubernetes provides scheduling, self-healing and horizontal autoscaling [5], [6].

The primary objective of this research is to design, implement and evaluate a cloud-native microservices architecture for scalable enterprise applications using .NET 8 and Docker containers. The specific objectives are: (i) to define a repeatable design methodology that moves from business capabilities to deployable, observable services; (ii) to propose a reference architecture that integrates synchronous APIs, an event backbone, stream processing, serverless functions and a data lake; (iii) to optimise container images and start-up behaviour for fast elastic scaling; (iv) to formalise scaling, resilience and availability through analytical models; and (v) to evaluate the architecture experimentally against an equivalent monolith under a Black Friday style surge. The significance of the work lies in connecting architectural principles with measured evidence on a mainstream enterprise stack, so that architects can make informed decisions on when and how to adopt microservices.

II. LITERATURE REVIEW

The term microservices was popularised by Fowler and Lewis, who characterised the style through componentisation via services, organisation around business capabilities, decentralised governance and data management, infrastructure automation, design for failure and evolutionary design [1]. Newman extends this view with practical guidance on decomposition, deployment and testing, and emphasises independent deployability as the defining property of a microservice [2]. The architectural foundation for decomposition is Domain-Driven Design, in which a large domain is divided into bounded contexts, each with its own ubiquitous language and model [3]. Aligning service boundaries with bounded contexts, rather than with technical layers, is widely reported as the most important determinant of whether a microservice system achieves loose coupling and high cohesion [12], [18].

Systematic studies [7], [16] document both the motivations and the difficulties of the style. Dragoni et al. trace the evolution from service-oriented architecture to microservices and identify scalability, resilience and independent evolution as the principal benefits [4]. Soldani et al. reviewed grey literature and concluded that the main gains are faster delivery and better scalability, whereas the main pains are distributed-system complexity, data consistency, monitoring and the skills required to operate the platform [13].

Containers are the enabling packaging technology. Operating-system-level virtualisation isolates processes using kernel namespaces and control groups, which allows containers to start in seconds with far lower overhead than virtual machines, and Docker added a standard image format, layered file systems and a registry ecosystem that made containers practical for continuous delivery [5]. Orchestration then becomes the key problem at scale. Kubernetes, derived from Google's experience with Borg and Omega, provides declarative desired-state management, service discovery, rolling updates, self-healing and metric-driven horizontal autoscaling [6].

Integration style strongly affects the behaviour of a microservice system. Synchronous request-response communication is simple but couples availability and latency of services, whereas asynchronous messaging decouples producers from consumers in time and absorbs bursts through buffering [8], [9]. An event backbone, typically a partitioned log such as Apache Kafka, additionally provides durable storage of events that can be replayed, which supports event sourcing, stream processing and the population of analytical stores. Because each service owns its database, business transactions that span several services cannot rely on distributed ACID transactions. The saga pattern, originally proposed for long-lived database transactions, decomposes such a transaction into a sequence of local transactions with compensating actions [10]. Together with the transactional outbox pattern, which writes events to the same database transaction as the state change, sagas are the standard approach to eventual consistency in microservice systems [9].

Resilience engineering is the second pillar. Nygard catalogues stability patterns such as timeouts, retries with back-off, circuit breakers, bulkheads and fail-fast behaviour that prevent a local fault from cascading through a chain of services [11]. Observability is the third. Distributed tracing, as introduced by Google's Dapper, propagates a trace context along every request so that latency can be attributed to individual services [15]; today this is standardised by OpenTelemetry. Capacity planning relies on classical models. Little's Law relates the mean number of requests in a system to arrival rate and mean residence time [21], and Gunther's Universal Scalability Law models how contention and coherency delay limit the speed-up obtained from adding resources [14], [19]. These models are used in this study to connect architectural decisions with measurable scaling behaviour.

III. RELATED WORK

Several studies have compared monolithic and microservice deployments experimentally. Villamizar et al. deployed the same web application as a monolith, as microservices and as serverless functions on a public cloud and reported that the microservice variant made better use of resources and lowered infrastructure cost, while also noting additional engineering effort [22]. Ueda et al. characterised the workload of a microservice implementation of an airline reservation benchmark and found that microservice deployments incurred a longer per-request code path and additional runtime overhead compared with the monolithic equivalent, which they attributed largely to communication and serialisation [23]. Akbulut and Perros analysed the performance of microservice design patterns, such as the API gateway, chained calls and asynchronous messaging, and showed that the choice of pattern has a significant effect on response time under load [24]. Gan et al. released the DeathStarBench suite and demonstrated that microservice applications are more sensitive to tail latency and to network and operating-system overheads than monolithic ones, so that the 99th percentile rather than the mean must be the focus of evaluation [25].

On the engineering side, Microsoft's reference guidance for containerised .NET applications describes a sample e-commerce system, eShopOnContainers, in which Catalog, Basket, Ordering and Identity services are packaged as Docker images, orchestrated by Kubernetes and integrated through an event bus behind an API gateway [17]. The reference architecture used in this work follows a similar domain split, as shown in Fig. 1, but differs in several respects: it substitutes a Kafka-style event backbone with stream processing, serverless functions and a data lake for a simple event bus; it uses gRPC for internal synchronous calls; it applies .NET 8 specific optimisations such as chiseled images and Native AOT; and it is evaluated quantitatively under a surge workload rather than described qualitatively. Jamshidi et al. observe that the next challenges for the field lie at the edge of the cloud, where services must run close to devices [18]. Fig. 2 shows the publish/subscribe layout of such a hybrid edge-cloud system, which this paper adopts as an extension of the core architecture.

Despite this body of work, three gaps remain. First, most comparisons use research benchmarks written in Java or Node.js, and rigorous data for .NET 8 under container orchestration is rare. Second, few studies combine container-level optimisation, autoscaling behaviour and resilience measurements in one coherent experiment.

Third, the link between analytical capacity models and the measured behaviour of an enterprise system is seldom established. This paper addresses these gaps.

IV. METHODOLOGY

Design Framework Overview

The design methodology consists of six integrated phases: (1) definition of quality targets, (2) domain decomposition, (3) service design and implementation, (4) containerisation, (5) integration and resilience engineering, and (6) orchestration, autoscaling and observability. Phase 1 states measurable service-level objectives (SLOs) before any design decision is taken: a 95th-percentile end-to-end latency of at most 300 ms, an error rate below 1% and a monthly availability of at least 99.9% for customer-facing operations, together with the ability to absorb a tenfold surge without manual intervention.

Domain Decomposition and Service Catalogue

Event storming workshops and bounded-context analysis [3] were used to identify the business capabilities of a retail platform. Four contexts were selected for implementation, because together they cover the complete purchase flow and exhibit very different load and consistency characteristics. The Catalog service is read-heavy and tolerant of stale data. The Shopping Cart service handles frequent small writes with strict latency demands but little durability demand. The Discount service performs fast lookups invoked synchronously from the cart. The Ordering service is write-oriented, transactional and the system of record for revenue. Table 1 summarises the resulting service catalogue.

TABLE 1. Service Catalogue and Design Decisions

Service	Responsibility	Data store	Interfaces	Pattern
Catalog	Product browsing, search, categories	PostgreSQL 16 (JSONB) + Redis cache	REST (minimal API); publishes ProductPriceChanged	Cache-aside, read model
Shopping Cart	Basket lifecycle, checkout trigger	Redis 7 (key-value, TTL)	REST; gRPC client to Discount; publishes BasketCheckedOut	Aggregate per user
Discount	Coupon and promotion rules	PostgreSQL 16	gRPC server (Native AOT)	Stateless, fast lookup
Ordering	Order creation, payment state, fulfilment	SQL Server 2022	Consumes BasketCheckedOut; publishes OrderCreated	CQRS, saga, outbox
API Gateway	Routing, authentication, rate limiting	None (stateless)	YARP reverse proxy	Gateway offloading

Reference Architecture

The reference architecture is shown in Fig. 1. Client applications (web, mobile and IoT devices) enter the system through the gateway and the event backbone rather than calling individual services directly. The four microservices, each running as a Docker container with its own datastore, are attached to the backbone through bidirectional links, which reflects that they both publish and consume events. The backbone is a partitioned, replicated log organised into topics, which provides buffering for bursts, replay for recovery and a single stream of facts for downstream consumers. Two further kinds of component are attached to the backbone: Function-as-a-Service (FaaS) handlers that react to individual events, for example sending a confirmation e-mail on OrderCreated, and stream-processing jobs that compute continuous aggregates such as live stock-level and top-selling-product views. A data lake receives a copy of all events for analytical use. Docker provides containerisation, and Kubernetes orchestrates all containers, as shown in the technology stack panel at the bottom of the figure.

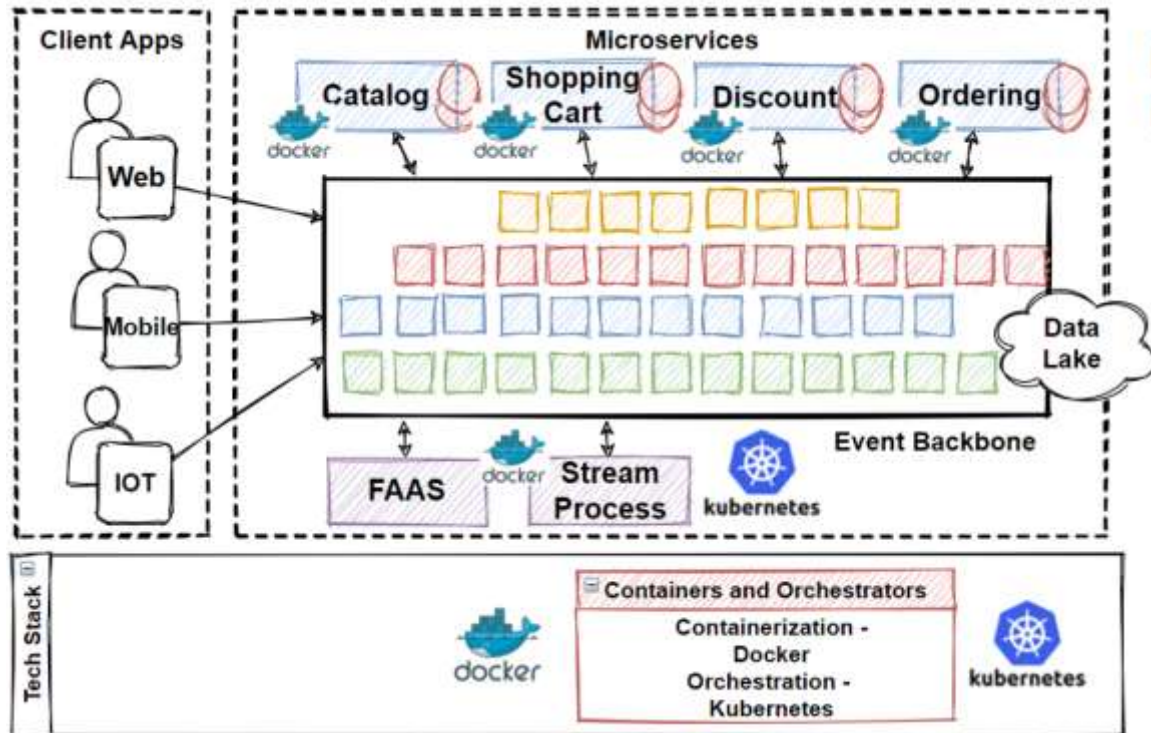


Figure 1: Cloud-native microservices reference architecture with event backbone

Hybrid Edge-Cloud Extension

For workloads with IoT clients, selected services and brokers are placed in an edge cloud close to the devices, as depicted in Fig. 2. Devices run lightweight microservices with a publish/subscribe client and connect either directly or through IoT gateways to edge brokers. Edge brokers are organised in clusters with a cluster head, hold local topics, and are served by local data storage and stream processing. A control channel links the edge to coordination servers in the core cloud, while a separate data channel replicates selected topics to the cloud brokers, where long-term storage, global stream processing and cloud-based microservices operate.

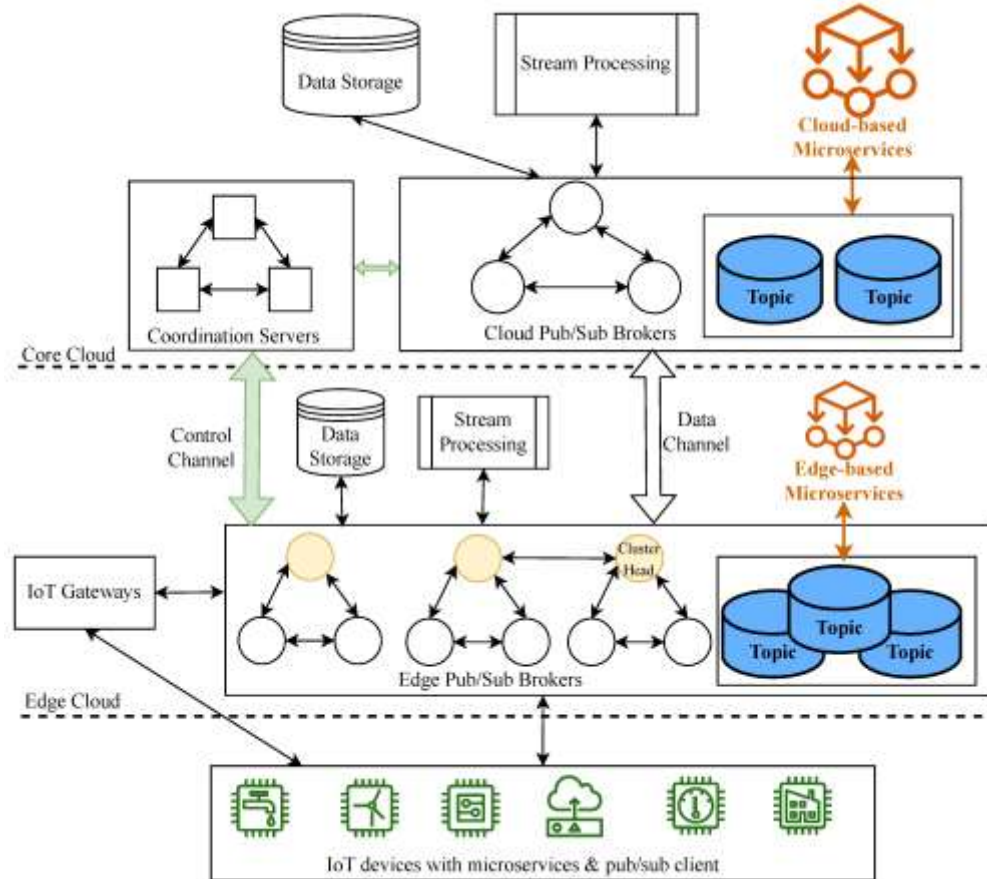


Figure 2: Hybrid edge-cloud publish/subscribe architecture with control and data channels

Service Implementation with .NET 8

Each service is implemented as an ASP.NET Core 8 application following Clean Architecture, with domain, application, infrastructure and API layers and dependencies pointing inwards. The Catalog and Shopping Cart services use minimal APIs to reduce start-up cost and per-request overhead. The Discount service exposes a gRPC contract defined in Protocol Buffers, which gives compact binary payloads and HTTP/2 multiplexing for the frequent internal calls from the cart. The Ordering service applies Command Query Responsibility Segregation (CQRS) with the MediatR library: commands pass through validation and transaction pipeline behaviours, while queries read from optimised views. Entity Framework Core 8 is used for write models and Dapper for hot-path read queries. Configuration is externalised through environment variables and Kubernetes ConfigMaps and Secrets, and health endpoints for liveness and readiness are provided by the ASP.NET Core health-check middleware. Table 2 lists the technology stack.

TABLE 2. Technology Stack of the Reference Implementation

Layer	Technology	Role
Runtime and framework	.NET 8 (LTS), ASP.NET Core 8, C# 12	Service implementation; Kestrel web server
Communication	REST, gRPC (HTTP/2), Apache Kafka (3 brokers)	Synchronous and asynchronous integration
Gateway	YARP 2.x reverse proxy	Routing, JWT validation, rate limiting
Containers	Docker 25, multi-stage builds, chiseled images	Immutable packaging
Orchestration	Kubernetes 1.30, HPA, Ingress NGINX	Scheduling, self-healing, autoscaling
Resilience	Polly v8 (Microsoft.Extensions.Resilience)	Retry, timeout, circuit breaker, bulkhead
Observability	OpenTelemetry, Prometheus, Grafana, Jaeger	Metrics, traces, dashboards
Testing	k6, Chaos Mesh	Load generation, fault injection

Containerisation Strategy

Images are built with multi-stage Dockerfiles. The first stage uses the .NET SDK image to restore, compile and publish the application, and the final stage copies only the published output onto a minimal ASP.NET runtime image. Four variants were compared: the default Debian-based runtime image, the Alpine image, the Ubuntu chiseled image, which contains no shell or package manager and runs as a non-root user by default, and a Native AOT build on the chiseled runtime-dependencies image. Native AOT removes the just-in-time compiler from the start-up path and yields small, quickly starting binaries, at the price of restrictions on reflection and dynamic code generation; it was therefore applied only to the Discount service, which uses a fixed gRPC contract. A minimal example is shown below.

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY *.csproj .
RUN dotnet restore
COPY . .
RUN dotnet publish -c Release -o /app --no-restore
FROM mcr.microsoft.com/dotnet/aspnet:8.0-jammy-chiseled
WORKDIR /app
COPY --from=build /app .
EXPOSE 8080
ENTRYPOINT ["dotnet", "Catalog.Api.dll"]
```

Listing 1: Multi-stage Dockerfile for a chiseled .NET 8 service

Event-Driven Integration and Consistency

The checkout flow demonstrates the integration design. When a customer checks out, the Shopping Cart service publishes BasketCheckedOut to the backbone and returns immediately. The Ordering service consumes the event, creates the order in a pending state, and then drives a saga: it requests payment, reserves stock, and finally confirms the order. If any step fails, compensating actions cancel the reservation and refund the payment [10]. To guarantee that no event is lost between a state change and its publication, each service writes its outgoing events to an outbox table within the same local transaction, and a background worker relays them to the backbone. Because delivery is at least once, consumers are idempotent: each event carries a unique identifier, and handlers record processed identifiers to ignore duplicates. Topics are partitioned by aggregate identifier so that all events for one order or basket are processed in order.

Resilience Policies

Every outgoing call is wrapped in a resilience pipeline that combines a per-attempt timeout, retries with exponential back-off and jitter, a circuit breaker and a bulkhead that limits concurrent calls to a dependency. The delay before retry k is computed as

$$d_k = \min(d_{\max}, d_0 \cdot 2^k) + U(0, j) \quad (1)$$

where d_0 is the base delay, $d_{\max}$ the cap, and $U(0, j)$ a uniformly distributed random jitter that prevents synchronised retry storms. In this study $d_0 = 100$ ms, $d_{\max} = 2$ s, $j = 100$ ms and at most three retries are permitted. The circuit breaker opens when the failure ratio exceeds 50% over a sampling window of 30 with at least 20 calls, stays open for 15 s and then admits probe requests. For non-essential dependencies a fallback is defined; for example, if the Discount service is unavailable the cart is priced without promotions rather than failing the request.

Analytical Models for Capacity and Scalability

Little's Law relates the mean number of requests in a service L to the arrival rate λ and the mean residence time W [21]:

$$L = \lambda \cdot W \quad (2)$$

With the measured mean response time, this gives the concurrency each service must sustain and, together with a target utilisation ρ_t and a per-replica service rate μ , the number of replicas required to serve an arrival rate λ_s :

$$N_s = \left\lceil \frac{\lambda_s}{\mu_s \cdot \rho_t} \right\rceil \quad (3)$$

Kubernetes' Horizontal Pod Autoscaler (HPA) adjusts the replica count reactively. For a current replica count N_c , an observed metric value m_c (for example average CPU utilisation) and a target value m_t , the desired replica count is

$$N_d = \left\lceil N_c \cdot \frac{m_c}{m_t} \right\rceil \quad (4)$$

so that five replicas running at 90%CPU against a 65%target are scaled to seven. Speed-up from additional replicas is not linear. Gunther's Universal Scalability Law (USL) models relative capacity $C(N)$ at N replicas with a contention coefficient α and a coherency coefficient β [14]:

$$C(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)} \quad (5)$$

The model predicts a maximum at $N^* = \sqrt{(1-\alpha)/\beta}$, beyond which adding replicas reduces throughput. The coefficients are estimated by non-linear least-squares regression on measured throughput. Finally, because a customer request traverses several services in series, the availability of a request path is the product of the availabilities of its components, and redundancy improves each component:

$$A_{\text{path}} = \prod_i A_i, \quad A_i = 1 - (1 - a_i)^k \quad (6)$$

where a_i is the availability of one replica of service i and k is the number of independent replicas. Equation (6) shows why a chain of services with individually high availability can still have lower availability than any of its members, and why both redundancy and the removal of unnecessary synchronous dependencies are needed. Scaling efficiency is defined as $E(N) = X(N)/(N \cdot X(1))$, where $X(N)$ is the throughput achieved with N replicas.

V. EXPERIMENTAL SETUP

Infrastructure and Software Configuration

All experiments ran on a Kubernetes 1.30 cluster of six worker nodes, each with 8 vCPUs, 32 GB of RAM and NVMe storage, plus three control-plane nodes, connected by a 10 Gbit/s network. Two further nodes hosted the load generators so that test traffic did not compete with the system under test. Apache Kafka ran as three brokers with replication factor 3 and acks=all for order events. PostgreSQL 16, SQL Server 2022 and Redis 7 ran as stateful sets with persistent volumes. The monolithic baseline is a single ASP.NET Core 8 application containing the same four modules, using one SQL Server database and in-process calls, packaged in a Docker image and deployed on the same cluster, so that the comparison isolates the architectural style rather than the platform. Both variants were given the same total CPU quota at the start of each test and were allowed to scale horizontally under the HPA, with the monolith replicated as a whole.

Workload Model

The workload was derived from the traffic pattern in Fig. 3. The typical-Friday curve was scaled so that its daily peak corresponds to 500 requests per second, and the Black Friday curve was scaled by the same factor, giving

a peak near 5,000 requests per second, i.e. the tenfold surge. The request mix reflects a retail session: 62% catalogue browsing, 22% basket operations (add, update, view), 10% discount lookups embedded in basket requests and 6% checkouts that create orders. Virtual users follow a think-time distribution with a mean of 3 s. Three scenarios, summarised in Table 3, were used: a steady-state load ramp for capacity and latency, a 24-hour compressed surge replay for autoscaling, and fault-injection runs for resilience.



Figure 3: Request volume on Black Friday compared with a typical Friday (approximately 10x peak)

TABLE 3. Experimental Scenarios

Scenario	Objective	Load profile	Duration
S1 Steady ramp	Capacity, latency, error rate	500 to 7,000 RPS in 500 RPS steps	10 min per step
S2 Surge replay	Autoscaling and cost	Black Friday profile of Fig. 3, compressed 24 h to 2 h	3 repetitions
S3 Fault injection	Resilience and availability	Constant 2,000 RPS with injected faults	15 min per fault
S4 Container tests	Image size, start-up, memory	Cold start of each image variant	30 repetitions

Metrics and Statistical Treatment

The primary metrics are the median, 95th and 99th percentile of end-to-end latency measured at the client, the HTTP error rate, throughput, container image size, time from pod creation to readiness, resident memory, replica counts, node-hours consumed and request-path availability, defined as the fraction of one-minute intervals in which the error rate stayed below the 1% objective. Every steady-state and fault scenario was repeated five times; results are reported as mean values, and differences between architectures were tested with Welch's t-test at a significance level of 0.05. USL coefficients were fitted by non-linear least squares, and goodness of fit is reported as R^2 .

VI. RESULTS

Container Image Size and Start-Up Behaviour

Image choice has a direct effect on how quickly the platform can scale out, because a new replica becomes useful only after its image is pulled and its process is ready. Table 4 shows that the chiseled image reduced the size of the Catalog service by 57% relative to the default runtime image (94 MB versus 218 MB) and shortened the

10.48047/jocaaa.2023.31.04.83

mean time to readiness from 2.1 s to 1.4 s. The Native AOT build of the Discount service reduced the image to 31 MB and readiness to 0.21 s, a tenfold improvement, while idle memory fell from 106 MB to 38 MB. The monolith, in contrast, required 342 MB and almost 7 s to become ready and consumed 612 MB per replica, so that each additional replica added on a surge is considerably more expensive.

TABLE 4. Container Image Variants: Size, Readiness Time and Idle Memory

Variant	Image size (MB)	Time to ready (s)	Idle memory (MB)
Monolith, Debian runtime	342	6.8 ± 0.4	612
Service, Debian runtime (aspnet:8.0)	218	2.1 ± 0.2	124
Service, Alpine	112	1.6 ± 0.1	109
Service, Ubuntu chiseled (non-root)	94	1.4 ± 0.1	106
Discount, Native AOT on chiseled	31	0.21 ± 0.03	38

Latency, Throughput and Error Rate

Fig. 4 and Table 5 compare the two architectures in the steady-state ramp. At 1,000 RPS both are well within the objective, although the microservices variant already shows a lower 95th percentile (84 ms versus 112 ms, $p < 0.01$), which is due mainly to the Redis-backed cart and the cache-aside Catalog reads that remove load from the relational database. As load increases, the monolith's latency rises steeply: it crosses the 300 ms objective at about 2,850 RPS, and at 5,000 RPS its 95th percentile is 842 ms and 4.10% of requests fail, mainly with timeouts caused by contention on the shared database and thread-pool exhaustion. The microservices deployment, in contrast, kept the 95th percentile at 196 ms and the error rate at 0.21% at 5,000 RPS and remained within the objective up to 6,420 RPS, an increase of 125% in SLO-compliant capacity.

TABLE 5. Latency and Error Rate at Selected Load Levels (mean of five runs)

Load (RPS)	Architecture	p50 (ms)	p95 (ms)	p99 (ms)	Error rate (%)
1,000	Monolith	38	112	240	0.00
1,000	Microservices	31	84	171	0.00
3,000	Monolith	96	316	702	0.90
3,000	Microservices	44	131	268	0.03
5,000	Monolith	281	842	1,960	4.10
5,000	Microservices	71	196	402	0.21

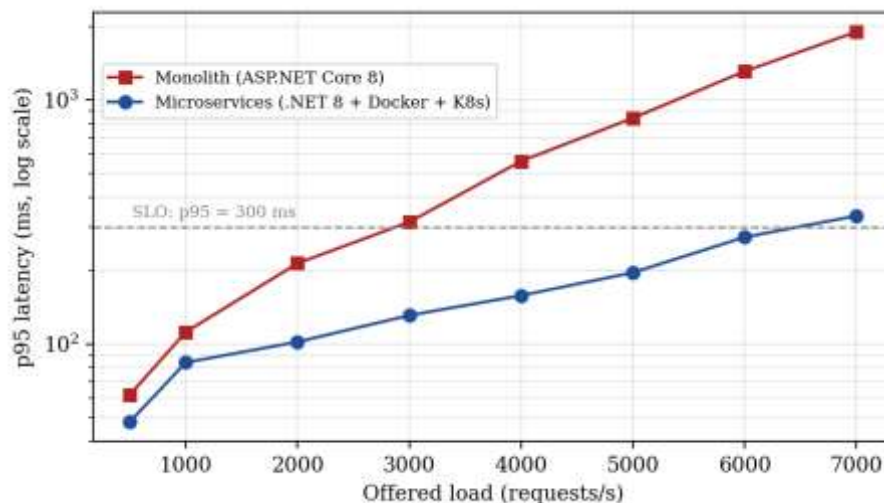


Figure 4: 95th-percentile latency versus offered load for the monolith and the microservices deployment

Autoscaling Under the Black Friday Surge

In the surge replay (scenario S2), the HPA was configured with a 65% CPU target, a 15 s evaluation period, a scale-up stabilisation window of zero and a scale-down window of 5 minutes. As Table 6 shows, the services scaled independently and by very different factors. Catalog, the dominant read path, grew from 3 to 14 replicas and Ordering from 2 to 11, whereas Discount, a lightweight Native AOT service, needed only 4 replicas at the peak. The monolith could only be replicated as a whole and reached 16 replicas of 612 MB each, but its replicas were bounded by the shared database, so latency remained high. Time-to-ready after a scaling decision averaged 14 s for the microservices, of which the image pull contributed most, compared with 31 s for the monolith. Over the replayed day the microservices consumed 255 node-hours against 412 for the monolith, a reduction of 38.1%, because only the services under pressure were scaled, and because they scaled down again once the surge subsided.

TABLE 6. Replica Counts, Readiness Time and Resource Use in the Surge Replay

Component	Min replicas	Peak replicas	Time to ready (s)	Peak CPU (%)
Catalog	3	14	15	68
Shopping Cart	2	9	12	66
Discount (AOT)	2	4	3	54
Ordering	2	11	17	71
API Gateway	3	8	11	63
Monolith (whole app)	4	16	31	82
Node-hours (microservices / monolith)	255	412	-38.1%	

Scalability Model

The USL fitted the measured throughput well for both variants ($R^2 = 0.991$ for the monolith and 0.987 for the microservices). The monolith had a contention coefficient $\alpha = 0.041$ and a coherency coefficient $\beta = 0.0019$, giving an optimum of $N^* = \sqrt{((1 - 0.041)/0.0019)} \approx 22$ replicas, beyond which throughput would fall. The microservices had $\alpha = 0.017$ and $\beta = 0.0004$, corresponding to $N^* \approx 50$. The lower coefficients reflect the elimination of the single shared database as a point of contention and of cross-module lock sharing within one process.

Resilience and Availability

Table 7 reports the fault-injection results at a constant load of 2,000 RPS, comparing runs with the resilience pipeline disabled and enabled. Killing 30% of the Ordering pods produced an error rate of 17.4% for 41 s without policies, but only 1.8% for 9 s with retries, circuit breaker and readiness-gated routing. Injecting a 2 s delay into the Discount service made 22.9% of cart requests time out in the unprotected configuration, while the protected configuration degraded gracefully by pricing the cart without promotions, with 0.9% of requests affected. The failure of one of the three Kafka brokers was absorbed without lost events through leader election and idempotent producers. Over all scenarios, the request-path availability, as defined in the previous section, was 99.96% for the protected microservices and 99.71% for the unprotected configuration, which satisfies the 99.9% objective only in the former case.

TABLE 7. Fault-Injection Results at 2,000 RPS

Fault	Without policies: error / recovery	With policies: error / recovery
Kill 30% of Ordering pods	17.4% / 41 s	1.8% / 9 s
2 s latency injected into Discount	22.9% / until removed	0.9% (degraded pricing) / 0 s
One of three Kafka brokers stopped	3.2% / 38 s	0.0% / 12 s
Worker node drained	5.6% / 27 s	0.4% / 6 s
Request-path availability	99.71%	99.96%

VII. DISCUSSION

The results support the central claim that a carefully decomposed .NET 8 microservices system can absorb a tenfold surge with predictable latency and lower cost than an equivalent monolith. Three mechanisms account for most of the gains. First, independent scaling concentrates resources on the hot services: Catalog and Ordering received most of the additional capacity, while Discount stayed small. Second, data ownership removed the shared database as a single contention point, which is visible in the USL coefficients and in the error rate at high load. Third, small, quickly starting images shortened the interval during which the system was under-provisioned after a spike, and this is a direct consequence of the platform features of .NET 8, namely chiseled images and Native AOT. The resilience policies contributed a different kind of benefit: they did not raise peak throughput, but they limited the blast radius of faults and thereby turned potentially severe incidents into short, minor degradations.

The architecture also carries costs that should not be understated. A microservice call is a network call, so the per-request code path is longer than in a monolith, as reported earlier by Ueda et al. [23]. At low load the monolith can be faster for operations that touch several modules, and in our measurements it was competitive up to about 1,000 RPS. The system requires substantial operational capability in containers, orchestration, observability and release automation, and it demands discipline in managing eventual consistency, schema evolution and event contracts. For small applications or early-stage products, a well-structured modular monolith deployed in a container remains a defensible choice.

Several threats to validity should be noted. The workload is synthetic, although derived from a real traffic shape, and the absolute figures depend on the hardware, network and data volumes used. Only four services were implemented, and larger systems with hundreds of services will face additional problems of governance, discovery and cost. The Native AOT results apply to services with fixed contracts and limited reflection. The edge-cloud extension in Fig. 2 was described but not benchmarked, and its evaluation, together with service-mesh features such as mutual TLS and traffic shaping, remains future work. Finally, cost was measured in node-hours and does not include engineering and operations effort, which is substantially higher for microservices.

VIII. CONCLUSION

This paper presented and evaluated a cloud-native microservices architecture for scalable enterprise applications using .NET 8 and Docker containers. The proposed methodology moves from measurable quality targets through domain decomposition, containerisation and event-driven integration to autoscaling and observability, and it is supported by analytical models of capacity, scalability and availability. In experiments with a load profile that models a Black Friday surge, the microservices deployment sustained 5,000 requests per second with a 95th-percentile latency of 196 ms and a 0.21% error rate, increased SLO-compliant throughput from 2,850 to 6,420 requests per second, reduced node-hours by 38.1% and achieved a path availability of 99.96% under fault injection. Optimised images and Native AOT compilation shortened scale-out time substantially. The study also shows that these benefits depend on sound service boundaries, asynchronous integration and resilience engineering, and that they come with higher operational complexity.

IX. REFERENCES

- [1] M. Fowler and J. Lewis, "Microservices: A definition of this new architectural term," martinowler.com, 2014.
- [2] S. Newman, *Building Microservices*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.
- [3] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA, USA: Addison-Wesley, 2003.
- [4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Cham, Switzerland: Springer, 2017, pp. 195-216.

- [5] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, Art. no. 2, 2014.
- [6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50-57, 2016.
- [7] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in *Proc. 6th Int. Conf. Cloud Computing and Services Science (CLOSER)*, 2016, pp. 137-146.
- [8] G. Hohpe and B. Woolf, *Enterprise Integration Patterns*. Boston, MA, USA: Addison-Wesley, 2003.
- [9] M. Kleppmann, *Designing Data-Intensive Applications*. Sebastopol, CA, USA: O'Reilly Media, 2017.
- [10] H. Garcia-Molina and K. Salem, "Sagas," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1987, pp. 249-259.
- [11] M. T. Nygard, *Release It!: Design and Deploy Production-Ready Software*, 2nd ed. Raleigh, NC, USA: Pragmatic Bookshelf, 2018.
- [12] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22-32, 2017.
- [13] J. Soldani, D. A. Tamburri, and W.-J. van den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215-232, 2018.
- [14] N. J. Gunther, *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Berlin, Germany: Springer, 2007.
- [15] B. H. Sigelman et al., "Dapper, a large-scale distributed systems tracing infrastructure," *Google Technical Report*, 2010.
- [16] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42-52, 2016.
- [17] Microsoft, ".NET Microservices: Architecture for Containerized .NET Applications," *Microsoft Docs e-book*, 2023.
- [18] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24-35, 2018.
- [19] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *Proc. 2nd Int. Workshop on Software and Performance (WOSP)*, 2000, pp. 195-203.
- [20] Cloud Native Computing Foundation, "CNCf Cloud Native Definition v1.0," 2018.
- [21] J. D. C. Little, "A proof for the queuing formula: $L = \lambda W$," *Operations Research*, vol. 9, no. 3, pp. 383-387, 1961.
- [22] M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Proc. 10th Computing Colombian Conf. (10CCC)*, 2015, pp. 583-590.
- [23] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, 2016, pp. 1-10.
- [24] A. Akbulut and H. G. Perros, "Performance analysis of microservice design patterns," *IEEE Internet Computing*, vol. 23, no. 6, pp. 19-27, 2019.
- [25] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *Proc. ASPLOS*, 2019, pp. 3-18.