Big Data — Stage 3 Project

Distributed Ingestion, Indexing and Search Pipeline

Adrian Budzich, Martyna Chmielińska

January 2026

Abstract

This report describes the Stage 3 Big Data project: a distributed pipeline that downloads books from Project Gutenberg, ingests them into a datalake, indexes the content into an inverted index, and provides a TF–IDF ranked search API. The system is implemented as containerized microservices orchestrated via Docker Compose, using ActiveMQ for asynchronous event delivery, Hazelcast for distributed state (index and search), and Nginx as a load balancer for horizontally scaled search replicas. We present the architecture, key implementation choices, fault tolerance behavior, and benchmarking results for three dataset sizes under different numbers of search replicas.

Contents

1 Project goal and requirements

The goal of Stage 3 is to deliver a distributed end-to-end pipeline with:

- **Ingestion service** that downloads and parses a book, stores it in a local datalake structure, and publishes events for the next stage.
- **Indexing service** that consumes ingestion events, builds an inverted index, and persists the indexed data.
- **Search service** that provides a search API with TF–IDF ranking and optional filters.
- **Scalability and resilience**: the system must be able to run on a single machine and across multiple machines, and tolerate nodes joining/leaving during execution.
- **Benchmarking**: measure ingestion throughput, indexing progress, and search latency/throughput for multiple dataset sizes.

2 Project repository

GitHub repository: Big Data Stage 3 repository

3 System architecture

3.1 High-level components

The solution consists of three Java 17 microservices and two supporting components:

- **Ingestion service (port 7001)**: downloads raw Gutenberg text, parses it into `header/body/meta`, stores the output in the datalake and publishes an event for indexing.
- **Indexing service (port 7002)**: consumes ingestion events from ActiveMQ and indexes documents into a distributed inverted index using Hazelcast.
- **Search service (port 7003)**: serves HTTP search requests and reads the index from Hazelcast to compute TF–IDF ranking.
- **ActiveMQ (port 61616)**: message broker used for asynchronous ingestion → indexing communication.
- **Nginx load balancer (port 18080)**: routes `/search` requests to the available search replicas.

3.2 Persistent storage: datalake volume

The ingestion service stores downloaded (and parsed) books in a **datalake** implemented as a Docker **named volume**. This keeps the data **outside of the container filesystem**, so containers can be removed and recreated without losing the downloaded dataset.

- Volume name: `<compose-project>_ingestion_datalake` (e.g. `task3_ingestion_datalake`).
- Mount point inside the container: `/app/datalake`.
- The volume is deleted only if the user runs `docker compose down -v`.

The ingestion service writes each book into a dated folder structure:

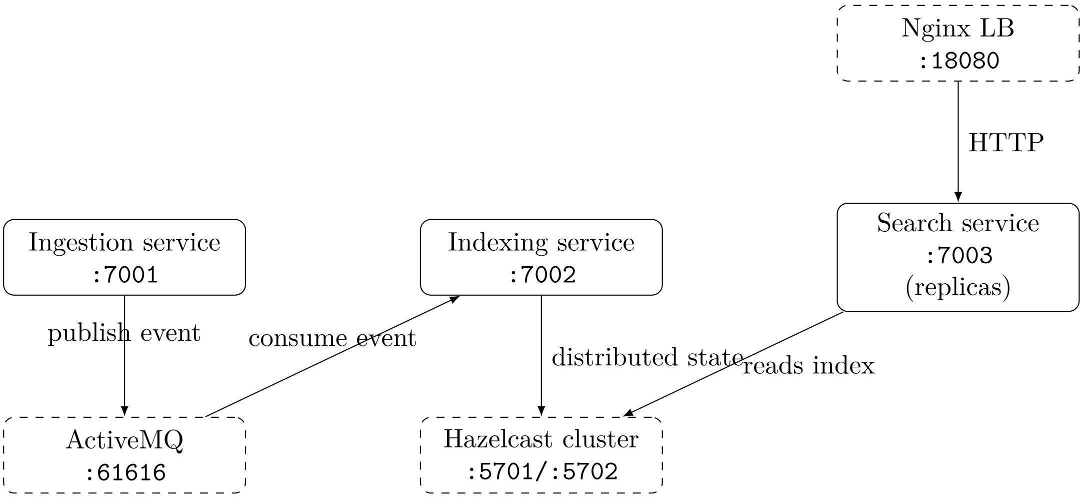

Figure 1: Architecture overview: ingestion publishes indexing events via ActiveMQ; indexing and search share a Hazelcast distributed state; Nginx load balances search replicas.

```
/app/datalake/YYYYMMDD/HH/<book_id>_header.txt
/app/datalake/YYYYMMDD/HH/<book_id>_body.txt
/app/datalake/YYYYMMDD/HH/<book_id>_meta.json
```

To inspect the datalake volume and verify persistence:

```
docker volume ls
# find the datalake volume
docker volume inspect task3\_ingestion\_datalake
# list contents via a temporary container
docker run --rm -v task3\_ingestion\_datalake:/datalake alpine ls -la /
    datalake
```

The index is a **derived** dataset (it can always be rebuilt from the datalake). In the implementation, indexing is triggered asynchronously by ingestion events via ActiveMQ.

3.3 Data flow

1. A client triggers ingestion for a Gutenberg `book_id` via `POST /ingest/{id}`.

2. Ingestion downloads the raw text, parses it and writes to the datalake folder structure.

3. Ingestion publishes an `ingestion.ingested` event to ActiveMQ.

4. Indexing consumes events, reads the datalake document, builds the inverted index, and persists the datamart.

5. Search executes TF–IDF ranking on the indexed data and returns results.

4 Deployment modes

4.1 Single-machine mode

In local mode, all services run on one machine using Docker Compose. Search can be scaled horizontally using `-scale search=N`. A helper PowerShell script `scripts/run-cluster.ps1` starts the stack and generates the Nginx upstream config automatically.

4.2 Multi-machine (cluster) mode

Cluster mode allows starting services on multiple machines in the same LAN. Each node runs its own containers, while Hazelcast forms a multi-member cluster and ActiveMQ can be hosted on one node. The system supports different partitions, e.g. running ingestion on machine A and indexing/search on machine B, then adding another search node dynamically.

4.3 Ports

Component	Container port	Host port (default)
Ingestion	7001	7001
Indexing	7002	7002
Search	7003	7003 (or behind LB)
Hazelcast (indexing)	5701	5701
Hazelcast (search)	5702	5702
ActiveMQ	61616	61616
Nginx LB	80	18080

5 Key implementation decisions

5.1 Asynchronous indexing with ActiveMQ

Ingestion publishes an event after writing a document to the datalake. This decouples ingestion from indexing and improves throughput because downloads and parsing do not block indexing work.

5.2 Distributed index and search with Hazelcast

Hazelcast is used as a shared, distributed state for:

- The inverted index (term → postings list),
- Document metadata required by ranking,
- Search-side access to the same indexed data.

Replication factor is configurable in the ingestion service, enabling consistent replication of content across nodes.

5.3 Load balancing of search replicas

Nginx routes incoming `/search` requests using `least_conn` strategy, distributing load across all available search instances. When search is scaled (e.g. 1 → 3 replicas), throughput increases and request latency decreases.

6 Benchmark methodology

6.1 Datasets

Three dataset sizes were used:

- **SMALL**: 10 documents
- **MEDIUM**: 50 documents
- **LARGE**: 155 documents (154 successfully ingested)

Document IDs were taken from the Project Gutenberg “Top downloads” page and de-duplicated.

6.2 Experimental factors

- **Search scale**: $\{1, 3\}$ search replicas.
- **Workload**: 200 search requests per dataset size.

6.3 Metrics

- **Ingestion throughput**: documents per second during `POST /ingest` calls.
- **Indexing completion**: indexed documents, vocabulary size (terms) and the remaining time after ingestion until the index becomes stable.
- **Search performance**: QPS and latency distribution (average, p50, p95, maximum).
- **Resource usage snapshots**: CPU and memory usage from `docker stats -no-stream` collected at key phases.

6.4 Benchmark runner

Benchmarks were executed using an automated PowerShell script `benchmarks/run-bench.ps1`. The script:

1. Starts the full stack in local mode.
2. Ingests a dataset.
3. Waits for indexing stability via `/index/status`.
4. Runs a search load test through the load balancer.
5. Captures summaries and resource snapshots into a timestamped run directory.

7 Benchmark results

Pipeline throughput (ingestion + indexing)

Table 1: Pipeline performance: ingestion throughput and indexing completion (Stage 3 benchmark run).

Scale	Pack	Ingest OK	Total	Ingest time [s]	Ingest [docs/s]	Index docs	Total pipeline [s]
1	SMALL	10	10	10.8	0.926	10	31.8
1	MEDIUM	49	50	54.7	0.896	49	126.7
1	LARGE	154	155	120.7	1.276	154	297.7
3	SMALL	10	10	9.6	1.039	10	21.6
3	MEDIUM	49	50	42.9	1.141	49	55.9
3	LARGE	154	155	125.4	1.228	154	138.4

Search performance (load balancer)

Table 2: Search benchmark results (load balancer at port 18080).

Scale	Pack	Requests	Total [s]	QPS	Avg [s]	P50 [s]	P95 [s]
1	SMALL	200	9.34	21.41	0.0277	0.0265	0.0368
1	MEDIUM	200	23.30	8.59	0.0974	0.0961	0.1141
1	LARGE	200	56.06	3.57	0.2605	0.2546	0.3192
3	SMALL	200	5.24	38.17	0.0069	0.0060	0.0090
3	MEDIUM	200	5.26	38.04	0.0071	0.0062	0.0078
3	LARGE	200	5.26	38.00	0.0072	0.0062	0.0088

7.1 Search throughput improvement when scaling

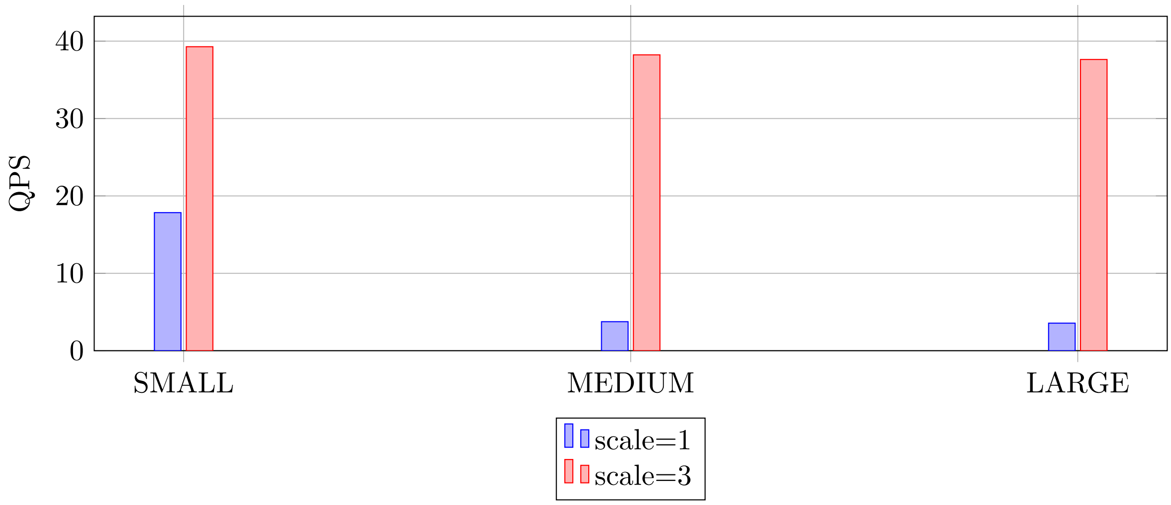

Figure 2: Search throughput (QPS) for different dataset sizes and number of search replicas.

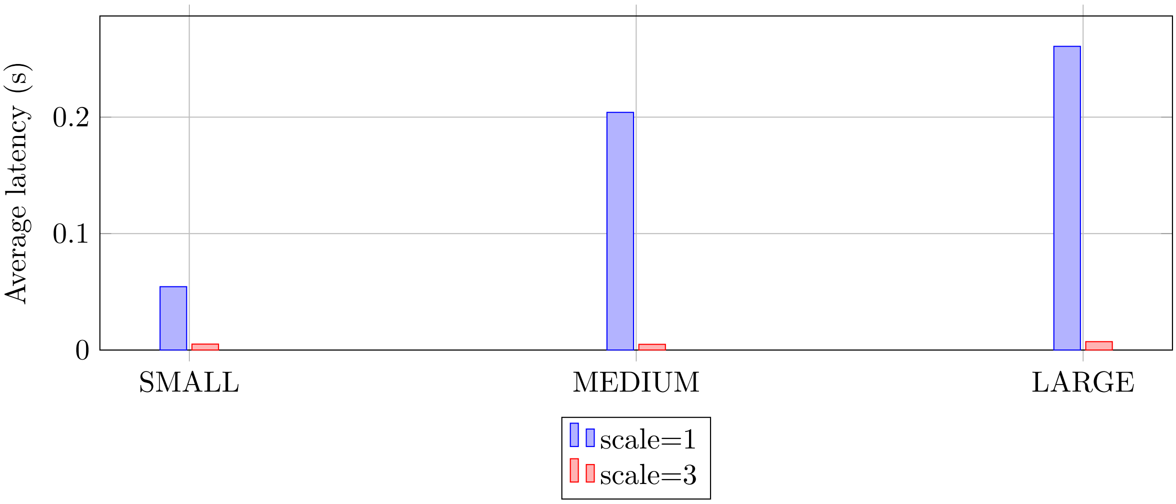

Figure 3: Average search latency for different dataset sizes and number of search replicas.

7.2 Discussion

The results show that adding search replicas significantly improves throughput. For MEDIUM and LARGE datasets, scaling from 1 to 3 replicas increased throughput from around 3.6–3.8 QPS to roughly 38 QPS. Latency also decreased accordingly.

Indexing time is measured as *additional waiting time after ingestion completes.* Because indexing runs concurrently with ingestion, large datasets may already be partially indexed by the time ingestion finishes.

8 Reproducibility

8.1 Starting the system locally

```
powershell -NoProfile -ExecutionPolicy Bypass -File .\scripts\run-cluster.ps1 -Mode local -ScaleSearch 1
# Load balancer: http://127.0.0.1:18080/status
```

8.2 Running benchmarks

Example benchmark configuration:

```
powershell -NoProfile -ExecutionPolicy Bypass -File .\benchmarks\run-bench.ps1 \
  -SmallN 10 -MediumN 50 -LargeN 155 \
  -ScaleList 1,3 \
  -SearchRequests 200 \
  -IndexTimeoutSmall 600 -IndexTimeoutMedium 1200 -IndexTimeoutLarge 2400
```

Each execution creates a folder in `benchmarks/runs/RUN_YYYYMMDD_HHMMSS` containing all CSV/JSON summaries.

9 Video demonstration

Video demonstration

YouTube: Watch the demo on YouTube

QR code:

This section documents the exact scenario used in the final video recording. The goal is to show that the system:

- runs on **two physical machines** on the same LAN,
- answers requests through a **load balancer**,
- continues operating when a service **leaves** and then **rejoins** the cluster,
- keeps the **datalake** isolated as persistent storage outside containers.

Nodes used in the demo

- Node A: `192.168.1.151` (runs LB + ActiveMQ + full stack)
- Node B: `192.168.1.139` (joins the cluster)

Step 1: Show node configuration (IP addresses)

```
:: Node A
ipconfig

:: Node B
ipconfig
```

Step 2: Start the cluster (CMD)

```
:: Node A (with infrastructure)
powershell -NoProfile -ExecutionPolicy Bypass -File .\scripts\run-cluster.ps1 -Mode cluster -Nodes "192.168.1.151,192.168.1.139" -Me 192.168.1.151 -Infra

:: Node B
powershell -NoProfile -ExecutionPolicy Bypass -File .\scripts\run-cluster.ps1 -Mode cluster -Nodes "192.168.1.151,192.168.1.139" -Me 192.168.1.139
```

Step 3: Verify that all services are reachable (local + remote)

```
:: Node A
docker ps
curl -i "http://127.0.0.1:18080/status"
curl -i "http://127.0.0.1:7001/status"
curl -i "http://127.0.0.1:7002/index/status"
curl -i "http://127.0.0.1:7003/hz/members"

curl -i "http://192.168.1.139:7001/status"
curl -i "http://192.168.1.139:7002/index/status"
curl -i "http://192.168.1.139:7003/hz/members"

:: Node B
docker ps
curl -i "http://127.0.0.1:7001/status"
curl -i "http://127.0.0.1:7002/index/status"
curl -i "http://127.0.0.1:7003/hz/members"

curl -i "http://192.168.1.151:18080/status"
curl -i "http://192.168.1.151:7001/status"
curl -i "http://192.168.1.151:7002/index/status"
curl -i "http://192.168.1.151:7003/hz/members"
```

Step 4: Ingest and automatically index a book

```
:: Node A
curl -s -X POST "http://127.0.0.1:7001/ingest/222"
curl -s "http://127.0.0.1:7002/index/status"
curl -i "http://127.0.0.1:18080/search?q=moon&limit=5"

:: Node B
curl -s "http://127.0.0.1:7002/index/status"
curl -i "http://127.0.0.1:7003/search?q=moon&limit=5"
```

Step 5: Query both via load balancer and directly

```
:: Node A (LB)
curl -i "http://127.0.0.1:18080/search?q=the&limit=3"
curl -i "http://127.0.0.1:18080/search?q=human&limit=3"
curl -i "http://127.0.0.1:18080/search?q=war&limit=3"

:: Node B (direct)
curl -i "http://127.0.0.1:7003/search?q=the&limit=3"
curl -i "http://127.0.0.1:7003/search?q=human&limit=3"
curl -i "http://127.0.0.1:7003/search?q=war&limit=3"
```

Step 6: Failure injection (stop search on Node B)

```
:: Node B
docker compose stop search
docker ps
curl -i "http://127.0.0.1:7003/hz/members"
curl -i "http://127.0.0.1:7003/search?q=moon&limit=5"

:: Node A (LB still works)
curl -i "http://127.0.0.1:7003/hz/members"
curl -i "http://127.0.0.1:18080/search?q=moon&limit=5"
```

Step 7: Recovery (start search again on Node B)

```
:: Node B
docker compose start search
docker ps
curl -i "http://127.0.0.1:7003/hz/members"
curl -i "http://127.0.0.1:7003/search?q=moon&limit=5"

:: Node A
curl -i "http://127.0.0.1:7003/hz/members"
```

Step 8: Stop indexing on Node A (search stays online)

```
:: Node A
docker compose stop indexing
docker ps
curl -i "http://127.0.0.1:7003/hz/members"
curl -i "http://127.0.0.1:18080/search?q=moon&limit=5"

:: Node B
curl -i "http://127.0.0.1:7003/hz/members"
curl -i "http://127.0.0.1:7003/search?q=moon&limit=5"
```

Step 9: Ingest while indexing is down, then recover indexing

```
:: Node A
curl -s -X POST "http://127.0.0.1:7001/ingest/34"
curl -i "http://127.0.0.1:7002/index/status"
curl -i "http://192.168.1.139:7002/index/status"
curl -i "http://127.0.0.1:18080/search?q=internet&limit=5"
```

```

:: Node B
curl -i "http://127.0.0.1:7002/index/status"
curl -i "http://127.0.0.1:7003/search?q=internet&limit=5"

:: Node A (restore indexing)
docker compose start indexing
docker ps
curl -s "http://127.0.0.1:7002/index/status"
curl -i "http://127.0.0.1:18080/search?q=internet&limit=5"
```

Step 10: Stop ingestion and indexing, keep searching existing data

```
:: Node A
docker compose stop ingestion
docker ps

:: Node B
docker compose stop indexing
docker ps
curl -i "http://127.0.0.1:7003/hz/members"

:: Ingest should fail now
curl -s -X POST "http://127.0.0.1:7001/ingest/999"

:: Search still works for already indexed data
curl -i "http://127.0.0.1:7003/search?q=commedia&limit=5"

:: Node A (LB query)
curl -i "http://127.0.0.1:18080/search?q=commedia&limit=5"

docker compose start ingestion
docker ps

:: Node B
docker compose start indexing
docker ps
```

10 Conclusion

Stage 3 delivers a distributed pipeline with decoupled ingestion and indexing, distributed indexing/search state, and horizontally scalable search behind a load balancer. Benchmarks confirm that scaling search replicas improves throughput and reduces latency under load. The architecture is reproducible and supports both single-machine and multi-machine deployments.

A Code excerpts

A.1 Cluster runner script

```
param(
  [ValidateSet("local","cluster")]
  [string]$Mode = "local",

  [string]$Nodes = "",

  [string]$Me = "",

  [string]$MqHost = "",

  [switch]$Infra,

  [int]$ScaleSearch = 1
)

$ErrorActionPreference = "Stop"

function Get-MyIp {
  try {
    $ip = (Get-NetIPAddress -AddressFamily IPv4 |
            Where-Object { $_.IPAddress -match "^\d+\.\d+\.\d+\.\d+$" -and
                $_.IPAddress -ne "127.0.0.1" } |
            Select-Object -First 1 -ExpandProperty IPAddress)
    return $ip
  } catch { return "" }
}

$projectRoot = Join-Path $PSScriptRoot ".."
$runtimeDir  = Join-Path $projectRoot ".runtime"
$nginxDir    = Join-Path $runtimeDir "nginx"
New-Item -ItemType Directory -Force -Path $nginxDir | Out-Null

$nginxConfPath = Join-Path $nginxDir "default.conf"
$overridePath  = Join-Path $runtimeDir "cluster.override.yml"

if ($Mode -eq "local") {

  $nginxText = @"
resolver␣127.0.0.11␣ipv6=off␣valid=2s;

upstream␣search_cluster␣{
␣␣zone␣search_cluster␣64k;
␣␣least_conn;
␣␣server␣search:7003␣resolve;
}

server␣{
␣␣listen␣80;

␣␣location␣/␣{
␣␣␣␣proxy_pass␣http://search_cluster;
␣␣␣␣add_header␣X-Upstream␣$upstream_addr␣always;
␣␣␣␣proxy_next_upstream␣error␣timeout␣http_502␣http_503␣http_504;
␣␣}
}
"@
  [System.IO.File]::WriteAllText($nginxConfPath, $nginxText, [System.Text.
     Encoding]::ASCII)

  Push-Location $projectRoot
  try {
    docker compose -f docker-compose.infra.yml -f docker-compose.yml up -d
        --build --scale search=$ScaleSearch
  } finally {
    Pop-Location
  }

  Write-Host "OK␣(LOCAL).␣LB:␣http://localhost:18080/status"
  exit 0
}

$nodeList = $Nodes.Split(",") | ForEach-Object { $_.Trim() } | Where-Object { $_ -ne "" }
if ($nodeList.Count -lt 1) { throw "Cluster␣mode␣requires␣-Nodes,␣e.g.␣-Nodes␣‘"192.168.1.144,192.168.1.139‘"" }

if ([string]::IsNullOrWhiteSpace(${Me})) { ${Me} = Get-MyIp }
if ([string]::IsNullOrWhiteSpace(${Me})) { throw "Cannot␣detect␣local␣IP.␣Pass␣-Me␣<your_ip>." }

if ([string]::IsNullOrWhiteSpace($MqHost)) { $MqHost = $nodeList[0] }
$mqUrl = "tcp://${MqHost}:61616"

$replFactor = [Math]::Min($nodeList.Count, 3)

$members = @()
foreach ($n in $nodeList) {
  $members += "${n}:5701"
  $members += "${n}:5702"
}
$membersCsv = ($members -join ",")

# upstreamy: lokalny search po docker DNS + zdalne po IP
$upstreams = @("␣␣server␣search:7003␣resolve␣max_fails=2␣fail_timeout=2s;")

foreach ($n in $nodeList) {
  if ($n -ne $Me) {
    $upstreams += "␣␣server␣${n}:7003␣max_fails=2␣fail_timeout=2s;"
  }
}

$nginxText = @"
resolver␣127.0.0.11␣ipv6=off␣valid=2s;

upstream␣search_cluster␣{
␣␣zone␣search_cluster␣64k;
␣␣least_conn;
$($upstreams␣-join␣"‘n")
}

server␣{
␣␣listen␣80;

␣␣location␣/␣{
␣␣␣␣proxy_pass␣http://search_cluster;
␣␣␣␣add_header␣X-Upstream␣‘$upstream_addr␣always;

␣␣␣␣proxy_connect_timeout␣3s;
␣␣␣␣proxy_send_timeout␣30s;
␣␣␣␣proxy_read_timeout␣30s;

␣␣␣␣proxy_next_upstream␣error␣timeout␣http_502␣http_503␣http_504;
␣␣}
}
"@
```

```
[System.IO.File]::WriteAllText($nginxConfPath, $nginxText, [System.Text.Encoding]::ASCII)

$override = @"
services:
␣␣ingestion:
␣␣␣␣ports:
␣␣␣␣␣␣-␣"7001:7001"
␣␣␣␣command:
␣␣␣␣␣␣-␣java
␣␣␣␣␣␣-␣-jar
␣␣␣␣␣␣-␣/app/app.jar
␣␣␣␣␣␣-␣--port=7001
␣␣␣␣␣␣-␣--mq=$mqUrl
␣␣␣␣␣␣-␣--indexingQueue=ingestion.ingested
␣␣␣␣␣␣-␣--origin=http://${Me}:7001
␣␣␣␣␣␣-␣--replFactor=$replFactor

␣␣indexing:
␣␣␣␣ports:
␣␣␣␣␣␣-␣"7002:7002"
␣␣␣␣␣␣-␣"5701:5701"
␣␣␣␣command:
␣␣␣␣␣␣-␣java
␣␣␣␣␣␣-␣-jar
␣␣␣␣␣␣-␣/app/app.jar
␣␣␣␣␣␣-␣--port=7002
␣␣␣␣␣␣-␣--mq=$mqUrl
␣␣␣␣␣␣-␣--ingestion=http://ingestion:7001
␣␣␣␣␣␣-␣--hzCluster=bd-hz
␣␣␣␣␣␣-␣--hzMembers=$membersCsv
␣␣␣␣␣␣-␣--hzPort=5701
␣␣␣␣␣␣-␣--hzInterface=${Me}

␣␣search:
␣␣␣␣ports:
␣␣␣␣␣␣-␣"7003:7003"
␣␣␣␣␣␣-␣"5702:5702"
␣␣␣␣command:
␣␣␣␣␣␣-␣java
␣␣␣␣␣␣-␣-jar
␣␣␣␣␣␣-␣/app/app.jar
␣␣␣␣␣␣-␣--port=7003
␣␣␣␣␣␣-␣--hzCluster=bd-hz
␣␣␣␣␣␣-␣--hzMembers=$membersCsv
␣␣␣␣␣␣-␣--hzPort=5702
␣␣␣␣␣␣-␣--hzInterface=${Me}
"@
$utf8NoBom = New-Object System.Text.UTF8Encoding($false)
[System.IO.File]::WriteAllText($overridePath, $override, $utf8NoBom)

Push-Location $projectRoot
try {
  if ($Infra) {
    docker compose -f docker-compose.infra.yml up -d --build
  }
  docker compose -f docker-compose.yml -f ./.runtime/cluster.override.yml up -d --build
} finally {
  Pop-Location
}

Write-Host "OK (CLUSTER). ME=${Me} MQ=$mqUrl R=$replFactor"
if ($Infra) {
  Write-Host "LB: http://${Me}:18080/status"
}
```

A.2 Benchmark runner script

```
param(
    # Pass as: -ScaleList 1,3   (real int array, not a string)
    [int[]]$ScaleList = @(1,3),

    # Dataset sizes = number of SUCCESSFULLY ingested documents
    [int]$SmallN  = 10,
    [int]$MediumN = 50,
    [int]$LargeN  = 200,

    # Search benchmark
    [int]$SearchRequests = 200,

    # Index timeouts (seconds)
    [int]$IndexTimeoutSmall  = 1200,
    [int]$IndexTimeoutMedium = 2400,
    [int]$IndexTimeoutLarge  = 5400,

    # Gutenberg candidates pool size (more candidates = less risk of "bad" ids)
    [int]$GutenbergPool = 600
)

$ErrorActionPreference = "Stop"

# IMPORTANT: in some PS builds, native stderr text can become "errors".
# We keep running unless a command really fails (non-zero exit code).
if (Get-Variable -Name PSNativeCommandUseErrorActionPreference -Scope Global -ErrorAction SilentlyContinue) {
    $global:PSNativeCommandUseErrorActionPreference = $false
}

$BenchRoot   = $PSScriptRoot
$ProjectRoot = (Resolve-Path (Join-Path $BenchRoot "..")).Path
$RunsRoot    = Join-Path $BenchRoot "runs"
New-Item -ItemType Directory -Force $RunsRoot | Out-Null

$RunDir = Join-Path $RunsRoot ("RUN_" + (Get-Date -Format "yyyyMMdd_HHmmss"))
New-Item -ItemType Directory -Force $RunDir | Out-Null

Write-Host ""
Write-Host "========================================"
Write-Host "BENCH RUN DIR: $RunDir"
Write-Host "PROJECT ROOT : $ProjectRoot"
Write-Host "========================================"
Write-Host ""

Set-Location $ProjectRoot

$CURL = "curl.exe"
```

```
if (-not (Get-Command $CURL -ErrorAction SilentlyContinue)) {
    throw "curl.exe␣not␣found.␣On␣Windows␣it␣should␣exist.␣If␣not:␣install␣curl␣or␣use␣Git␣Bash."
}

$INGEST = "http://127.0.0.1:7001"
$INDEX  = "http://127.0.0.1:7002"
$LB     = "http://127.0.0.1:18080"

function Curl-HttpCode([string]$Url) {
    try {
        return (& $CURL -s -o NUL -w '%{http_code}' $Url)
    } catch {
        return ""
    }
}

function Wait-Http200([string]$Url, [int]$TimeoutSec = 180) {
    $sw = [System.Diagnostics.Stopwatch]::StartNew()
    while ($sw.Elapsed.TotalSeconds -lt $TimeoutSec) {
        $code = Curl-HttpCode $Url
        if ($code -eq "200") { return $true }
        Start-Sleep -Seconds 2
    }
    return $false
}

function Exec-CmdToLog([string]$CmdLine, [string]$LogPath) {
    # Avoid PowerShell pipeline "NativeCommandError" noise.
    cmd.exe /c "$CmdLine␣>␣\"$LogPath\"␣2>&1"
    return $LASTEXITCODE
}

function Compose-Down() {
    Write-Host "`n>>>␣DOCKER␣COMPOSE␣DOWN␣(-v)"
    $log = Join-Path $RunDir "compose_down.log"
    $code = Exec-CmdToLog "docker␣compose␣-f␣docker-compose.infra.yml␣-f␣docker-compose.yml␣down␣-v␣--remove-orphans" $log
    if ($code -ne 0) {
        Write-Host "WARNING:␣compose␣down␣exit=$code␣(see␣compose_down.log)"
    }
}

function Start-Local([int]$ScaleSearch) {
    Compose-Down

    Write-Host "`n>>>␣START␣LOCAL␣(search␣replicas␣=␣$ScaleSearch)"
    $log = Join-Path $RunDir ("start_local_S" + $ScaleSearch + ".log")

    $cmd = "powershell␣-NoProfile␣-ExecutionPolicy␣Bypass␣-File␣\"$ProjectRoot\\scripts\\run-cluster.ps1\"␣-Mode␣local␣-ScaleSearch␣$ScaleSearch"
    $code = Exec-CmdToLog $cmd $log
    if ($code -ne 0) {
        throw "run-cluster.ps1␣failed␣(exit=$code).␣Check␣$log"
    }

    if (-not (Wait-Http200 "$LB/status" 240)) {
        throw "LB␣not␣ready␣at␣$LB/status.␣Check:␣docker␣logs␣task3-lb-1"
    }
    if (-not (Wait-Http200 "$INDEX/index/status" 240)) {
        throw "Indexing␣not␣ready␣at␣$INDEX/index/status"
    }

    docker ps --format "table␣{{.Names}}\t{{.Status}}\t{{.Ports}}" |
        Out-File (Join-Path $RunDir ("docker_ps_S" + $ScaleSearch + ".txt")) -Encoding utf8

    Write-Host "OK␣(LOCAL).␣LB:␣$LB/status"
}

function Get-GutenbergTopIds([int]$Max = 200) {
    try {
        $resp = Invoke-WebRequest -Uri "https://www.gutenberg.org/browse/scores/top" -UseBasicParsing -TimeoutSec 25
        $html = $resp.Content
        $matches = [regex]::Matches($html, '/ebooks/(\d+)')
        $ids = @()
        foreach ($m in $matches) { $ids += [int]$m.Groups[1].Value }
        $ids = $ids | Select-Object -Unique
        if ($ids.Count -gt $Max) { $ids = $ids[0..($Max-1)] }
        return $ids
    } catch {
        return @()
    }
}

# Candidate pool (with fallback)
$TOP_IDS = Get-GutenbergTopIds $GutenbergPool
if ($TOP_IDS.Count -lt 250) {
    # Fallback list (extend it freely)
    $TOP_IDS = @(
        1342,84,11,1661,2701,98,1952,1080,2600,174,46,76,120,1400,2542,4363,3207,4300

        6130,408,141,43,160,55,219,996,25,1951,158,844,254,1260,2148,2000,730,816,10,

        2591,2500,844,1524,16328,28054,9830,345,1184,158,2680,64317,514,514,

        2554,2397,260,1260,2680,45,1232,30254,74,1184,20203,215,33,36,41
    ) | Select-Object -Unique
}

# Some Gutenberg ids are "bad" for our downloader (no plain text / moved / redirects).
# This is EXACTLY what caused your MEDIUM to stop at 49/50 earlier (id=89).
$BLACKLIST = @(
    89
)
$TOP_IDS = $TOP_IDS | Where-Object { $BLACKLIST -notcontains $_ }

function Snapshot-DockerStats([string]$Tag, [int]$ScaleSearch) {
    $out = Join-Path $RunDir ("S0_docker_stats_" + $Tag + "_S" + $ScaleSearch + ".csv")
    "#␣name,cpu,mem_usage,mem_perc" | Out-File $out -Encoding utf8
    docker stats --no-stream --format "{{.Name}},{{.CPUPerc}},{{.MemUsage}},{{.MemPerc}}" |
        Add-Content -Path $out -Encoding utf8
}
```

```
function Get-IndexStatus() {
    try { return Invoke-RestMethod -Uri "$INDEX/index/status" -TimeoutSec 5 } catch { return $null }
}

function Wait-IndexingStable([string]$PackName, [int]$TimeoutSec, [int]$ScaleSearch) {
    $poll  = Join-Path $RunDir ("K2_index_" + $PackName + "_S" + $ScaleSearch + "_poll.csv")
    $final = Join-Path $RunDir ("K2_index_" + $PackName + "_S" + $ScaleSearch + "_final.json")

    Write-Host "`n>>> WAIT INDEXING (STABLE): $PackName timeout=${TimeoutSec}s [scale=$ScaleSearch]"
    "ts,elapsed_s,docs,terms" | Out-File $poll -Encoding utf8

    $sw = [System.Diagnostics.Stopwatch]::StartNew()
    $stable = 0
    $prevDocs = -1
    $prevTerms = -1

    while ($sw.Elapsed.TotalSeconds -lt $TimeoutSec) {
        $st = Get-IndexStatus
        if ($null -eq $st) {
            Add-Content -Path $poll -Value "$(Get-Date -Format s),$([int]$sw.Elapsed.TotalSeconds),NA,NA" -Encoding utf8
            Start-Sleep -Seconds 3
            continue
        }

        $docs  = [int]$st.docs
        $terms = [int]$st.terms

        Add-Content -Path $poll -Value "$(Get-Date -Format s),$([int]$sw.Elapsed.TotalSeconds),$docs,$terms" -Encoding utf8
        Write-Host ("  t={0,6}s  docs={1,5}  terms={2,8}  stable={3}" -f ([int]$sw.Elapsed.TotalSeconds), $docs, $terms, $stable)

        if ($docs -eq $prevDocs -and $terms -eq $prevTerms) {
            $stable++
        } else {
            $stable = 0
        }

        if ($stable -ge 3) {
            $st | ConvertTo-Json -Depth 8 | Out-File $final -Encoding utf8
            Write-Host "INDEXING STABLE: docs=$docs terms=$terms"
            return @{ ok=$true; wait_s=[int]$sw.Elapsed.TotalSeconds; docs=$docs; terms=$terms }
        }

        $prevDocs = $docs
        $prevTerms = $terms
        Start-Sleep -Seconds 3
    }

    $last = Get-IndexStatus
    if ($last) { $last | ConvertTo-Json -Depth 8 | Out-File $final -Encoding utf8 }
    Write-Host "INDEXING␣TIMEOUT␣(saved␣last␣status)"
    return @{ ok=$false; wait_s=[int]$sw.Elapsed.TotalSeconds; docs=($last.docs); terms=($last.terms) }
}

function Ingest-Pack([string]$PackName, [int]$TargetOkDocs, [int]$ScaleSearch) {
    $jsonl   = Join-Path $RunDir ("K1_ingest_" + $PackName + "_S" + $ScaleSearch + ".jsonl")
    $summary = Join-Path $RunDir ("K1_ingest_" + $PackName + "_S" + $ScaleSearch + "_summary.txt")
    $idsFile = Join-Path $RunDir ("DATASET_" + $PackName + "_ids.txt")

    Write-Host "‘n>>>␣INGEST:␣$PackName␣target_ok=$TargetOkDocs␣[scale=$ScaleSearch]"
    "#␣$(Get-Date␣-Format␣’yyyy-MM-dd␣HH:mm:ss’)" | Out-File $jsonl -Encoding utf8

    $ok = 0
    $used = New-Object System.Collections.Generic.List[int]
    $sw = [System.Diagnostics.Stopwatch]::StartNew()

    foreach ($id in $TOP_IDS) {
        if ($ok -ge $TargetOkDocs) { break }

        $url  = "$INGEST/ingest/$id"
        $resp = & $CURL -s --connect-timeout 3 --max-time 240 -X POST $url
        $resp | Add-Content -Path $jsonl -Encoding utf8

        $good = $false
        try {
            $obj = $resp | ConvertFrom-Json
            if ($obj.http_status -eq 200) { $good = $true }
        } catch {
            $good = $false
        }

        if ($good) {
            $ok++
            $used.Add([int]$id)
        } else {
            Write-Host "␣␣skip␣id=$id␣(not␣ok)" -ForegroundColor DarkYellow
        }
    }

    $sw.Stop()

    if ($ok -lt $TargetOkDocs) {
        Write-Host "WARNING:␣only␣$ok/$TargetOkDocs␣ingested␣OK.␣Consider␣increasing␣-GutenbergPool." -ForegroundColor Yellow
    }

    $used | Out-File $idsFile -Encoding utf8

    $rate = if ($sw.Elapsed.TotalSeconds -gt 0) { [Math]::Round(($ok / $sw.Elapsed.TotalSeconds), 4) } else { 0 }
    @(
        "pack=$PackName␣docs_ok=$ok␣docs_target=$TargetOkDocs␣time_s=$([Math]::Round($sw.Elapsed.TotalSeconds,3))␣docs_per_s=$rate",
        "ids_file=$([IO.Path]::GetFileName($idsFile))"
    ) | Out-File $summary -Encoding utf8

    Write-Host "OK␣docs:␣$ok␣/␣$TargetOkDocs"

    return @{ ok=$ok; used=$used.ToArray(); time_s=[Math]::Round($sw.Elapsed.TotalSeconds,3); rate=$rate }
}

function Percentile([double[]]$Arr, [double]$P) {
    if ($Arr.Count -eq 0) { return 0 }
    $sorted = $Arr | Sort-Object
    $idx = [int][Math]::Floor(($P / 100.0) * ($sorted.Count - 1))
    return $sorted[$idx]
}

function Run-SearchBenchmark([string]$PackName, [int]$N, [int]$ScaleSearch) {
    $csv = Join-Path $RunDir ("K3_search_" + $PackName + "_S" + $ScaleSearch + "_latency.csv")
    $sum = Join-Path $RunDir ("K3_search_" + $PackName + "_S" + $ScaleSearch + "_summary.txt")

    $terms = @('love','war','peace','world','time','man','woman','night','king','day','good','great','death','life','heart','friend','fire','sea','home','money')

    Write-Host "`n>>>␣SEARCH␣BENCH:␣$PackName␣requests=$N␣[scale=$ScaleSearch]"
    "iter,term,latency_s,http_code" | Out-File $csv -Encoding utf8

    # Warmup
    for ($w=0; $w -lt 10; $w++) {
        $q = Get-Random -InputObject $terms
        $u = "$LB/search?q=$([uri]::EscapeDataString($q))&limit=5"
        & $CURL -s -o NUL --max-time 30 $u | Out-Null
    }

    $lat = New-Object System.Collections.Generic.List[double]

    $sw = [System.Diagnostics.Stopwatch]::StartNew()
    for ($i=1; $i -le $N; $i++) {
        $q = Get-Random -InputObject $terms
        $url = "$LB/search?q=$([uri]::EscapeDataString($q))&limit=5"

        # ONE curl call that returns both time_total and http_code
        $out = & $CURL -s -o NUL -w '%{time_total}␣%{http_code}' --max-time 30 $url
        $parts = $out -split '␣'

        $t = 0.0
        [double]::TryParse($parts[0], [ref]$t) | Out-Null
        $codeStr = if ($parts.Count -ge 2) { $parts[1] } else { "" }

        $lat.Add($t)
        "$i,$q,$t,$codeStr" | Add-Content -Path $csv -Encoding utf8
    }
    $sw.Stop()

    $total = $sw.Elapsed.TotalSeconds
    $qps = if ($total -gt 0) { [Math]::Round(($N / $total), 4) } else { 0 }

    $avg = if ($lat.Count -gt 0) { [Math]::Round(($lat | Measure-Object -Average).Average, 4) } else { 0 }
    $p50 = [Math]::Round((Percentile $lat.ToArray() 50), 4)
    $p95 = [Math]::Round((Percentile $lat.ToArray() 95), 4)
    $mx  = [Math]::Round(($lat | Measure-Object -Maximum).Maximum, 4)

    @(
        "pack=$PackName scale=$ScaleSearch",
        "requests=$N total_s=$([Math]::Round($total,3)) qps=$qps",
        "latency_s avg=$avg p50=$p50 p95=$p95 max=$mx"
    ) | Out-File $sum -Encoding utf8

    Write-Host "DONE qps=$qps avg=$avg p50=$p50 p95=$p95"

    return @{ qps=$qps; avg=$avg; p50=$p50; p95=$p95; max=$mx; total_s=[Math]::Round($total,3) }
}

$summaryCsv = Join-Path $RunDir "SUMMARY_full.csv"
"scale,pack,target_ok,ingest_ok,ingest_time_s,ingest_rate_docs_s,index_ok,index_wait_s,index_docs,index_terms,search_requests,qps,lat_avg_s,lat_p95_s" |
    Out-File $summaryCsv -Encoding utf8

foreach ($scale in $ScaleList) {

    # SMALL
    Start-Local $scale
    Snapshot-DockerStats "START_SMALL" $scale
    $ingSmall = Ingest-Pack "SMALL" $SmallN $scale
    $idxSmall = Wait-IndexingStable "SMALL" $IndexTimeoutSmall $scale
    Snapshot-DockerStats "AFTER_INDEX_SMALL" $scale
    $srchSmall = Run-SearchBenchmark "SMALL" $SearchRequests $scale
    Snapshot-DockerStats "AFTER_SEARCH_SMALL" $scale
    "$scale,SMALL,$SmallN,$($ingSmall.ok),$($ingSmall.time_s),$($ingSmall.rate),$($idxSmall.ok),$($idxSmall.wait_s),$($idxSmall.docs),$($idxSmall.terms),$SearchRequests,$($srchSmall.qps),$($srchSmall.avg),$($srchSmall.p95)" |
        Add-Content -Path $summaryCsv -Encoding utf8

    # MEDIUM
    Start-Local $scale
    Snapshot-DockerStats "START_MEDIUM" $scale
    $ingMed = Ingest-Pack "MEDIUM" $MediumN $scale
    $idxMed = Wait-IndexingStable "MEDIUM" $IndexTimeoutMedium $scale
    Snapshot-DockerStats "AFTER_INDEX_MEDIUM" $scale
    $srchMed = Run-SearchBenchmark "MEDIUM" $SearchRequests $scale
    Snapshot-DockerStats "AFTER_SEARCH_MEDIUM" $scale
    "$scale,MEDIUM,$MediumN,$($ingMed.ok),$($ingMed.time_s),$($ingMed.rate),$($idxMed.ok),$($idxMed.wait_s),$($idxMed.docs),$($idxMed.terms),$SearchRequests,$($srchMed.qps),$($srchMed.avg),$($srchMed.p95)" |
        Add-Content -Path $summaryCsv -Encoding utf8

    # LARGE
    Start-Local $scale
    Snapshot-DockerStats "START_LARGE" $scale
    $ingLarge = Ingest-Pack "LARGE" $LargeN $scale
    $idxLarge = Wait-IndexingStable "LARGE" $IndexTimeoutLarge $scale
    Snapshot-DockerStats "AFTER_INDEX_LARGE" $scale
    $srchLarge = Run-SearchBenchmark "LARGE" $SearchRequests $scale
    Snapshot-DockerStats "AFTER_SEARCH_LARGE" $scale
    "$scale,LARGE,$LargeN,$($ingLarge.ok),$($ingLarge.time_s),$($ingLarge.rate),$($idxLarge.ok),$($idxLarge.wait_s),$($idxLarge.docs),$($idxLarge.terms),$SearchRequests,$($srchLarge.qps),$($srchLarge.avg),$($srchLarge.p95)" |
        Add-Content -Path $summaryCsv -Encoding utf8
}

Write-Host ""
Write-Host "========================================"
Write-Host "ALL␣DONE.␣Results␣are␣in:␣$RunDir"
Write-Host "SUMMARY:␣$summaryCsv"
Write-Host "========================================"
Write-Host ""
```