

Five containers, one cloud: **Mezarpoly** on Docker & Azure

A real-time multiplayer board game we built and designed as five containers: run locally with Docker Compose and deployed as five separate components on Azure Container Apps.

Azim

PART 1

Gloria

PART 2

Kutay

PART 3

mezarpoly.fun

1

Slide 1 — Five containers,one cloud:Mezarpoly on Docker & Azure

[Azim] Good morning. We are Azim, Gloria and Kutay. Our project is Mezarpoly, a browser-based multiplayer property-trading game that the three of us built and that real players use at mezarpoly.fun. We designed it from the start as a system of five containers, and today we show that system running locally and on Microsoft Azure.

In the next ten minutes we will show the five containers, how they talk to each other synchronously and asynchronously, how every one of them runs as a separate component on Microsoft Azure, and what went wrong on the way. I will start with the application and Docker, Gloria continues with communication, data and the cloud platform, and Kutay finishes with the deployment, troubleshooting and the proof.

The application and Docker

Azim
PRESENTER

- What Mezarpoly is and what the course asks for
- The architecture: five containers, two networks
- The containers and how their images are built

Slide 2 — The application and Docker

(Divider slide — Azim starts Part 1: "The application and Docker".)

Three parts, three presenters

1 · Azim

The application and the requirements ·
the architecture · the containers and
their images

2 · Gloria

Docker Compose · the synchronous
path · Kafka and the worker ·
PostgreSQL · why Azure Container
Apps · the Azure architecture

3 · Kutay

The deploy script · security decisions ·
problems and fixes · proof that it works
· cost and next steps · questions

Slide 3 — Three parts, three presenters

[Azim] The talk has three parts. I cover the application and Docker: what the game is, what the course asks for, the architecture and how the images are built. Gloria then explains how the containers talk to each other, the database, and why we chose Azure Container Apps. Kutay closes with how we deployed it, the security decisions, the problems we solved, and the evidence that it runs end to end.

Mezarpoly

Our game: real-time, browser-based, 2-8 players, live at mezarpoly.fun



- Roll dice, buy cities, build houses, trade, auction
- Live game over a **WebSocket**
- Accounts, shop, friends, leaderboards, ranked league
- Designed from day one as **five services**, switched on by configuration
- Public site: a light single-server mode was enough
- This project: the full **five-container** design

Mezarpoly · Docker & Azure

4

Slide 4 — MezarpolyOur game: real-time, browser-based, 2–8 players, live at mezarpoly.fun

Mezarpoly is the game we built. It is not a toy example: it has a rules engine, rooms, bots, accounts, a shop, friends, leaderboards and a ranked league, and people actually play it.

We designed it from the beginning as five services: a web front, the game server, a database, a message broker and a worker. Each part is switched on by configuration. When we put the public site online we found that, at its current size, the full set was not needed, so mezarpoly.fun runs the same code in a light single-server mode. For this course we run the full five-container design exactly as planned. This screenshot was taken from the Azure deployment.

Grade map

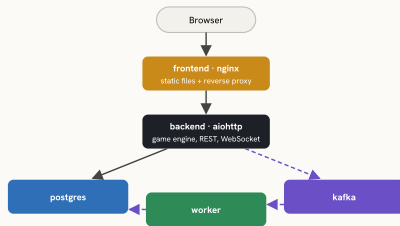
Grade	Requirement	How it is met
3.0	Application starts with docker-compose.yml	<code>docker compose up --build</code> starts all five services
4.0	Two containers communicate over a Docker network	frontend (nginx) proxies HTTP + WebSocket to backend on network <code>web</code>
4.5	Third container: a database	PostgreSQL 17 on network <code>data</code> : accounts, results, shop, friends
5.0	Fourth container for asynchronous messaging	Kafka + worker: game events → Kafka → worker → PostgreSQL
6.0	All containers in the cloud as separate components	Azure Container Apps: one Container App per container

Course goal: package code and dependencies into containers that run the same in every environment, and build a small microservice-style system from them. Each grade's *result* is shown live: grades 3.0-5.0 on slide 11, grade 6.0 on slide 22.

Slide 5 — Grade map

The course goal is to package an application and its dependencies into containers so that it runs the same everywhere, and to build a small system of cooperating services. This table maps every grade step to the part of the project that satisfies it. The project targets the highest step, 6.0: every container runs on Azure as its own component. Each lower step is a building block for the next one, so the rest of the talk follows this table from top to bottom.

Five containers, two networks



Network web
frontend, backend

Network data
backend, postgres, kafka, worker

Solid arrows: synchronous request/response.
Dashed arrows: asynchronous events.
The internet-facing frontend has **no route** to the database.

Slide 6 — Five containers, two networks

Here is the whole system on one slide. The browser only ever talks to the frontend, an nginx container. nginx serves the static game files itself and forwards the API, authentication and WebSocket traffic to the backend, which is the Python game server.

The backend writes to PostgreSQL synchronously and publishes game events to Kafka asynchronously. A separate worker reads those events and stores them in PostgreSQL.

There are two Docker networks. The frontend is only on the web network, so even if someone broke into nginx they could not reach the database directly. The backend is the only bridge between the two networks. This is network segmentation, the network version of least privilege.

What runs where

Service	Image	Port	Role
frontend	own: Node 22 build → nginx 1.29	80	Static client, reverse proxy for /api /auth /ws
backend	own: Python 3.13 slim	8095	Rules engine, rooms, bots, accounts, REST, WebSocket, Kafka producer
postgres	postgres:17-alpine	5432	Relational database (19 tables)
kafka	apache/kafka:3.9.1 (KRaft)	9092	Message broker, topic <code>mezarpoly.events</code>
worker	own: Python 3.13 slim	—	Kafka consumer → PostgreSQL

Three images are built from this repository, two are official images. All four long-running services have health checks.

Slide 7 — What runs where

Three of the images are mine and two are official ones. Kafka runs in KRaft mode, which means it does not need a separate ZooKeeper container: one container is both broker and controller.

Every long-running service has a health check, which matters for the start-up order on the next slides.

Three good habits in the Dockerfiles

Multi-stage build

The frontend compiles the Vue templates in a Node 22 stage, then copies only the result into nginx. No Node.js in the final image: smaller and safer.

Non-root user

backend and worker run as user `app` (uid 10001), not root. A compromised process gets very few rights inside the container.

Layer caching

Dependencies are copied and installed before the source code, so a code change rebuilds in seconds instead of reinstalling every library.

```
HEALTHCHECK --interval=10s --timeout=3s --retries=5 \
  CMD python -c "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8095/healthz', timeout=2)"
```

Slide 8 — Three good habits in the Dockerfiles

Three habits we applied. First, a multi-stage build for the frontend: the build tools stay in the first stage, the final image is just nginx plus static files.

Second, the Python containers do not run as root. Third, the order of instructions is chosen for the layer cache: dependencies first, code last.

The health check at the bottom is how Docker knows the backend is really ready, not just started.

Handover: "So we have five containers and good images. Gloria will now show how Compose starts them in the right order and how they talk to each other."

Communication, data and the cloud platform

Gloria
PRESENTER

- Docker Compose and the start-up order
- The synchronous path through nginx
- The asynchronous path through Kafka
- PostgreSQL, why Azure Container Apps, the Azure architecture

Slide 9 — Communication, data and the cloud platform

(Divider slide — Gloria starts Part 2: "Communication, data and the cloud platform".)

One command, the right order

```
backend:
  depends_on:
    postgres: { condition: service_healthy }
    kafka: { condition: service_healthy }
  networks: [web, data]

frontend:
  depends_on:
    backend: { condition: service_healthy }
  networks: [web]

networks:
  web: {} # Frontend .. backend
  data: {} # backend .. postgres / kafka / worker
```

Start-up order

1. postgres + kafka until *healthy*
2. backend + worker
3. frontend, then it is reachable

Containers find each other by **service name** (postgres:5432) via Docker's internal DNS. Data lives in named volumes and survives `docker compose down`.

Slide 10 — One command, the right order

`depends_on` alone only orders the start; it does not wait for a service to be ready. With condition `service_healthy`, Compose waits until the health check passes. That gives a clean order: database and Kafka first, then backend and worker, then the frontend.

No container knows another container's IP address. They use service names, which Docker resolves inside each network.

It runs locally with one command

```
$ docker compose ps # 3.0
```

SERVICE	IMAGE	STATUS
backend	mezarpoly-backend	Up 41 seconds (healthy)
frontend	mezarpoly-frontend	Up 30 seconds 0.0.0.0:8080->80
kafka	apache/kafka:3.9.1	Up 4 minutes (healthy)
postgres	postgres:17-alpine	Up 4 minutes (healthy)
worker	mezarpoly-worker	Up 41 seconds

```
$ docker network inspect ... # 4.0 / 4.5
```

mezarpoly_web:	frontend-1, backend-1
mezarpoly_data:	backend-1, postgres-1, kafka-1, worker-1

```
$ kafka-consumer-groups.sh --describe # 5.0
```

GROUP	TOPIC	OFFSET	END	LAG
mezarpoly-stats	mezarpoly.events	2	2	0

```
$ psql -c "SELECT id, type FROM game_events"
```

id	type
1	game_started
2	game_finished

A real game was played at localhost:8080. Both events travelled backend → Kafka → worker → PostgreSQL; **LAG 0** means the worker has processed everything.

Slide 11 — It runs locally with one command

This slide shows the result the course asks for at each local grade step, captured from a real run on a laptop.

docker compose ps: one command started all five services, and four of them report healthy. That is grade 3.0. The network listing shows the frontend and backend share the web network, and the backend, database, Kafka and worker share the data network: the frontend talks to the backend, and the backend talks to the database. That is 4.0 and 4.5.

For 5.0: we played one short game. The Kafka consumer group shows two messages produced and two consumed, lag zero, and the game_events table contains exactly those two events, game started and game finished. So data really flows backend, Kafka, worker, database.

What happens when you press "Play now"

1. **nginx** serves HTML, JS, CSS from its own disk
2. The client opens **/ws**; nginx forwards the *Upgrade* headers to the backend
3. The backend creates the room, the bots and the board in memory and starts the game
4. Accounts, results and the shop are read and written in **PostgreSQL**

```
location = /ws {  
    proxy_pass $backend;  
    proxy_http_version 1.1;  
    proxy_set_header Upgrade $http_upgrade;  
    proxy_set_header Connection $connection_upgrade;  
    proxy_read_timeout 3600s;  
}
```

Slide 12 — What happens when you press "Play now"

The game itself runs over a WebSocket, a connection that stays open so the server can push updates to every player immediately.

A reverse proxy has to be told to pass the Upgrade and Connection headers, otherwise the WebSocket never opens. We also raised the read timeout to one hour: the default sixty seconds would cut a player who thinks for a minute.

Kafka and the worker

Producer (backend): fire and forget

When a match starts or ends, the backend puts a JSON event on topic `mezarpoly.events` and continues immediately. If Kafka is slow or down, the game never waits.

Consumer (worker): at-least-once

The worker writes the event to `game_events` and `event_counts` in one transaction and commits the Kafka offset afterwards. A crash never loses an event.

```
async for msg in consumer:
    async with db.transaction():
        INSERT INTO game_events ...
        INSERT INTO event_counts ...
        ON CONFLICT DO UPDATE n = n + 1
    await consumer.commit() # after the write

{"type": "game_finished", "room": "PHAEJ",
 "board": "classic", "turns": 1,
 "humans": 2, "winner": "DockerB"}
```

Slide 13 — Kafka and the worker

This is the requirement for grade 5.0. The backend is the producer. It sends the event in a background task, so the game loop never blocks on Kafka.

The worker is the consumer, in group `mezarpoly-stats`. The important detail is the order: it writes to the database first and only then commits the offset. If it crashed in between, Kafka would deliver the same message again. The cost is that an event could, rarely, be stored twice; we preferred that to losing events. This is called at-least-once delivery.

The status page reads the result back through the backend, so the data makes a full circle: frontend, backend, Kafka, worker, database, backend, frontend.

One data layer, two databases

We wrote the game's data layer once, against the small sqlite3 interface. A ~150-line **adapter** (`pgcompat.py`) speaks the same interface to PostgreSQL and translates each query.

In the container setup `MP_DATABASE_URL` selects PostgreSQL; the light single-server mode uses SQLite. Our whole test suite runs against PostgreSQL to prove it.

SQLite	PostgreSQL
<code>?</code>	<code>%s</code>
<code>INSERT OR IGNORE</code>	<code>ON CONFLICT DO NOTHING</code>
<code>AUTOINCREMENT</code>	<code>BIGSERIAL</code>
<code>MAX(a, b)</code>	<code>GREATEST(a, b)</code>
<code>lastrowid</code>	<code>RETURNING id</code>

Slide 14 — One data layer, two databases

In the five-container design the database lives in its own container, so it has to be a network database server: PostgreSQL. In the light single-server mode an embedded SQLite file is enough.

To keep one code base for both, we wrote the data layer once and added a thin adapter that behaves like Python's `sqlite3` module but talks to PostgreSQL and translates the dialect. The proof is that our automated tests pass against PostgreSQL.

Why Azure Container Apps

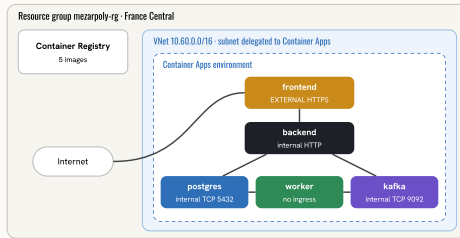
Option	Verdict
Virtual machine + Docker Compose	One machine, not "separate components"; we would maintain the OS
Azure Kubernetes Service	Powerful but heavy to learn and run for a small project
App Service	Great for web apps, not for TCP services like PostgreSQL and Kafka
Container Apps	Chosen: one app per container, internal/external ingress, TCP support, pay per use, Kubernetes managed by Azure

Slide 15 — Why Azure Container Apps

We compared four options. A virtual machine running Compose would work, but it is one machine, not separate components, and we would be patching an operating system. Kubernetes is the industry standard but too heavy here. App Service cannot run PostgreSQL and Kafka as containers.

Container Apps gives each container its own app, with internal and external ingress, TCP support and per-use pricing. Under the hood it is Kubernetes, but Microsoft runs it.

Only the frontend faces the internet



Registry: ACR Basic; postgres and kafka imported from Docker Hub.

VNet: needed because Container Apps only supports **TCP ingress** in a custom VNet.

Replicas: exactly one per app; game rooms live in the backend's memory.

Slide 16 — Only the frontend faces the internet

This is the cloud version of the architecture. Everything lives in one resource group in France Central, which is one of the few regions the student subscription allows.

PostgreSQL and Kafka speak plain TCP, not HTTP, and Container Apps only offers TCP ingress inside a custom virtual network, so we created a VNet with a subnet delegated to Container Apps and put the environment inside it.

The ingress types mirror the Docker networks: only the frontend is external; backend, postgres and kafka are internal; the worker has no ingress at all because it only makes outgoing connections.

Handover: "That is the target architecture in the cloud. Kutay will now show how we actually built it, what went wrong, and the proof that it works."

Deployment, security, troubleshooting, proof

Kutay
PRESENTER

- The deploy script, step by step
- Security decisions
- Problems we hit and how we fixed them
- Evidence that it works, cost and next steps

Slide 17 — Deployment, security, troubleshooting, proof

(Divider slide — Kutay starts Part 3: "Deployment, security, troubleshooting, proof".)

deploy.sh: seven idempotent steps

1. Resource group + container registry
2. VNet + subnet delegated to `Microsoft.App`
3. Container Apps environment in that subnet
4. postgres — internal TCP 5432, password as a secret
5. kafka — internal TCP 9092, single-node KRaft
6. worker (no ingress) + backend (internal HTTP)
7. frontend — external HTTPS, then tell the backend its public origin

```
az containerapp create -g mezarpoly-ng -n postgres \
  --environment mezarpoly-env \
  --registry-identity system --image $REG/postgres:17-
alpine \
  --secrets pgpass="$PG_PASS" \
  --env-vars POSTGRES_PASSWORD=secretref:pgpass \
  --ingress internal --transport tcp \
  --target-port 5432 --exposed-port 5432
```

Slide 18 — deploy.sh: seven idempotent steps

We did not click the environment together in the portal. One script built with the Azure CLI creates everything, and it is idempotent: running it again skips what exists and only rolls out new image tags. The script is the documentation of the environment, and anyone can reproduce it.

The example shows how the database app is created: image pulled with the app's managed identity, password stored as a secret, internal TCP ingress on port 5432.

Decisions that reduce risk

Small attack surface

One external endpoint (HTTPS, certificate by Azure). Database and Kafka are unreachable from the internet.

Secrets, not plain text

Random passwords generated at deploy time, stored as Container App secrets, injected with `secretref:`.

No registry password

Apps pull images with a system-assigned managed identity that has pull rights on the registry.

Least privilege

Non-root containers; the frontend has no network path to data.

CSRF protection

The backend accepts POST and WebSocket only from its own public origin.

Real client IP

nginx forwards the first `X-Forwarded-For` address so rate limits see the user, not the load balancer.

Slide 19 — Decisions that reduce risk

Six security decisions. The most important: there is exactly one public entry point. Secrets are never written into the script or the environment variables in plain text, and there is no registry password at all because each app authenticates with a managed identity.

The application keeps its existing protections too: origin checks against CSRF and per-IP rate limits, which needed the real client IP to be forwarded through two proxy layers.

The "404 page not found" puzzle

Symptom: the home page loaded, but `/healthz`, `/api` and the WebSocket returned `404 page not found`.

Diagnosis: that text is the environment's internal router (Envoy), not nginx. nginx forwarded to `http://backend`; internal HTTP traffic is routed by Host header and only the app's **full internal name** matches.

Fix:

```
BACKEND_URL=http://backend.internal.<env domain>,  
computed in deploy.sh so it never comes back.
```

Why did postgres:5432 work?

TCP ingress is forwarded directly, so the short name resolves. HTTP ingress goes through the router, which matches host names. Same environment, two behaviours.

Slide 20 — The "404 page not found" puzzle

This was our most instructive problem. After the first deployment the site loaded, so nginx was fine, but every call to the backend returned "404 page not found".

The wording gave it away: that is not nginx's error page, it is the internal Envoy router of the Container Apps environment. Locally, Docker's DNS turns "backend" into the container. In Azure, internal HTTP traffic is routed by host name, and only the full internal name matches. TCP apps such as postgres are reachable by the short name because TCP is forwarded directly.

We fixed it by computing the full name from the environment's default domain in the deploy script, so a fresh deployment works first time.

Other problems on the way

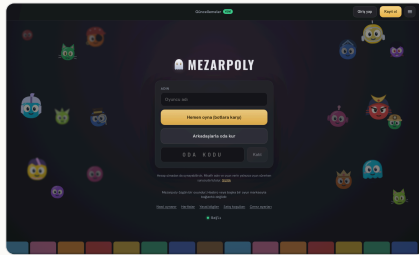
Problem	Cause	Fix
Container Apps commands rejected	Resource provider <code>Microsoft.App</code> not registered	<code>az provider register</code> , wait for <i>Registered</i>
Resources refused in some regions	Azure for Students region policy	France Central
No TCP ingress	Requires a custom VNet	VNet + delegated /23 subnet
Logins and moves rejected	Origin check did not know the new public URL	Set <code>MP_ORIGINS</code> after the frontend exists
<code>GroupCoordinatorNotAvailable</code> in worker logs	Kafka still creating internal topics	Harmless; retries until "listening"

Slide 21 — Other problems on the way

Smaller problems, each with a clear cause. Two are about the student subscription: the Container Apps provider had to be registered, which takes a few minutes, and only some regions are allowed.

One is about the application's own security: the origin check rejected requests until the backend learned its new public URL. The last one looks scary in the logs but is harmless: Kafka is still creating its internal topics, and the worker retries until it is listening.

It runs end to end on Azure



```
App      State      External  Transport
postgres Running   False     Tcp
kafka    Running   False     Tcp
worker   Running   False     Http
backend  Running   False     Http
frontend Running   True      Auto

[events] connected to Kafka: kafka:9092
[worker] listening: kafka:9092 / mezarpoly.events
[worker] processed: game_started 2RYA9
```

Slide 22 — It runs end to end on Azure

The evidence. Left: the game served from the Azure public URL. Right: the five Container Apps, only the frontend external, and the logs of a real game.

The last log line is the important one: a guest game in room 2RYA9 started, the backend published the event, and the worker processed it into PostgreSQL. That single line proves the asynchronous path and the separate cloud components work together.

Cost, and what we would do next

Cost control

- `istop.sh` deactivates every revision → 0 replicas, no compute cost
- `istart.sh` brings everything back in 1-2 minutes
- Registry (Basic) is the only fixed cost
- Data in postgres/kafka is ephemeral in this setup

Next steps

- Managed PostgreSQL + Event Hubs (Kafka API) for durable data
- CI/CD: GitHub Actions builds and deploys on every push
- Central logs and metrics with Log Analytics
- Shared room state (Redis) to scale the backend horizontally

Slide 23 — Cost, and what we would do next

Running five apps all day would burn student credit, so two scripts stop and start the whole system by deactivating and reactivating revisions. When stopped, nothing is billed for compute.

We are honest about the limits: data in the database and Kafka is not persistent in this setup. For production we would use Azure's managed PostgreSQL and Event Hubs, add a CI/CD pipeline, and move room state to Redis so the backend could scale to several replicas.

Thank you Questions?

- Five containers, two networks, one command locally
- Synchronous path through nginx; asynchronous events through Kafka
- Five separate Container Apps on Azure, one public endpoint
- Reproducible with one script; debugged with evidence

Slide 24 — Thank youQuestions?

To summarise: our game in its full five-container design, on two networks, started with one command; a synchronous path through nginx and an asynchronous event path through Kafka; and five separate Container Apps on Azure with a single public endpoint, built by one reproducible script.

Thank you. We are happy to take questions, and we can show the running system or the scripts live.