

Part 2:

Communication, data and the cloud platform

Your presenter guide: what to say on each of your slides, the background you need to answer questions confidently, likely questions with model answers, and a checklist for the day.

PRESENTER

Gloria

Your part at a glance

1 · Azim

Application, architecture,
containers, images · slides 1–8 · ~4
min

2 · Gloria

Compose, communication paths,
database, cloud platform · slides
9–16 · ~4 min

3 · Kutay

Deployment, security,
troubleshooting, proof, Q&A ·
slides 17–24 · ~4 min

You are in the middle of the talk. Azim has shown **what** the five containers are. You show **how they work together** — how Compose starts them, the two communication paths (synchronous through nginx, asynchronous through Kafka), the database — and then you take the audience to the cloud: why Azure Container Apps and what the Azure architecture looks like.

Slide	Topic	Time
9	Part 2 divider	0:05
10	Docker Compose: one command, the right order	0:40
11	Local evidence: grades 3.0–5.0 running	0:30
12	Synchronous path: what happens when you press "Play now"	0:40
13	Asynchronous path: Kafka and the worker	0:55
14	One data layer, two databases	0:35
15	Why Azure Container Apps	0:35
16	Azure architecture: only the frontend faces the internet	0:45
	Total	≈ 4:45

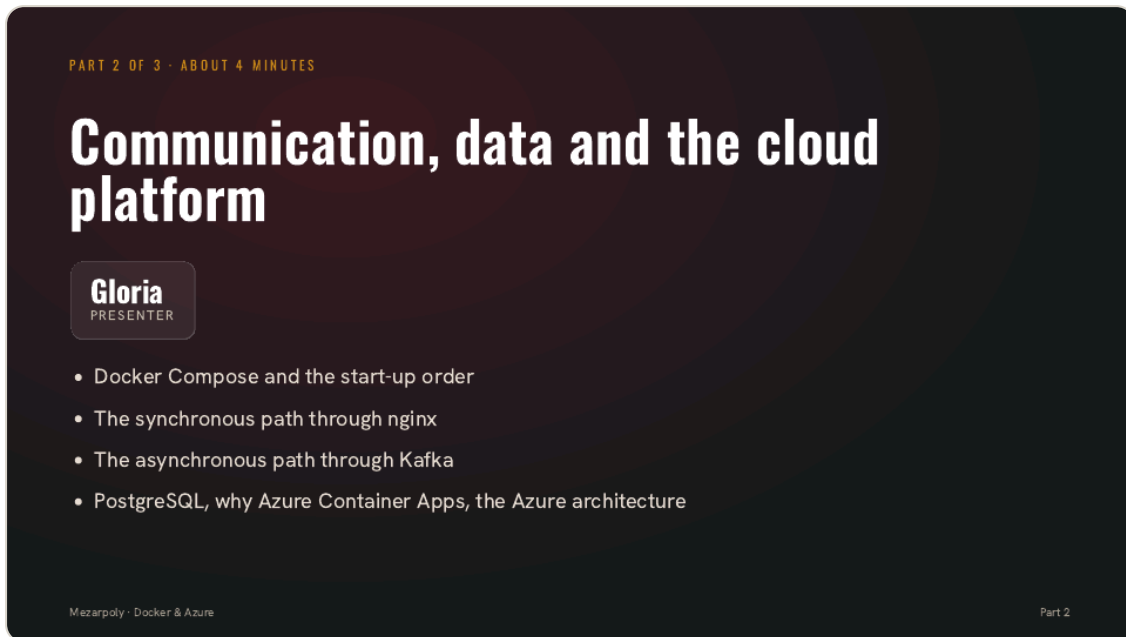
Your three key messages

- Compose starts everything in the **right order** using health checks, and containers find each other **by name**.
- There are **two paths**: the game waits for nothing slow; events go through **Kafka** and are never lost (**at-least-once**).
- **Container Apps** fits because each container becomes its own app, with internal/external ingress and TCP support.

How you take over and hand over

Azim ends with: "Gloria will now show how Compose starts them in the right order...". When Azim stops sharing, share your screen with Slides-Part2-Gloria.pdf already open full screen on slide 9, and start immediately. **Online:** each of you shares your own screen with your own slide file (Slides-Part... PDF, opened full screen before your turn). Say the handover sentence, then stop sharing; the next presenter starts sharing right away. Practise this switch once in a test call: it should take under five seconds. At the end, stop sharing so Kutay can share. At the end of slide 16 say: *"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."*

Part 2 divider — Communication, data and the cloud platform



WHAT TO SAY

Thank you, Azim. I will show how the containers work together, how data flows, and how we moved the system to the cloud.

REMEMBER

- Thank the previous speaker by name: it makes the talk feel like one team.

Docker Compose — one command, the right order

DOCKER COMPOSE

PART 2 · GLORIA

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`.

Mezarpoly · Docker & Azure

10

WHAT TO SAY

Docker Compose describes all five services in one file and starts them with one command:

```
docker compose up.
```

Order matters. The backend cannot start before the database and Kafka are ready. A plain `depends_on` only orders the start; it does not wait. So we use **condition: service_healthy**: Compose waits until the health check passes.

That gives a clean sequence: first PostgreSQL and Kafka until they are healthy, then the backend and the worker, and last the frontend, which is the moment the site becomes reachable.

No container knows another container's IP address. The backend simply connects to `postgres:5432` and Docker's internal DNS resolves the name. Data lives in named volumes, so it survives `docker compose down`.

REMEMBER

- `depends_on` alone = order only; `service_healthy` = wait until ready.
- Containers are addressed by service name.

Evidence — it runs locally with one command (grades 3.0–5.0)

EVIDENCE · GRADES 3.0 – 5.0

PART 2 · GLORIA

It runs locally with one command

```
$ docker compose ps
```

SERVICE	IMAGE	STATUS
backend	mezarpoly-backend	Up 41 seconds (healthy)
frontend	mezarpoly-frontend	Up 39 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 ...
```

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

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

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.

Mezarpoly · Docker & Azure
11

WHAT TO SAY

Before we follow the data, here is the proof that all of this really runs. These outputs were captured on a laptop after one `docker compose up`.

Top left: `docker compose ps`. All five services are up and four report **healthy**. That is grade 3.0: the application runs in Docker with one command.

Below: the two networks. Frontend and backend share **web**; backend, PostgreSQL, Kafka and the worker share **data**. So the frontend talks to the backend, and the backend talks to the database: grades 4.0 and 4.5.

On the right, grade 5.0. We played one short game. Kafka shows two messages written and two read, with **lag zero**, and the `game_events` table contains exactly those two events: game started and game finished. The data really flowed backend → Kafka → worker → database.

REMEMBER

- Point at each block and say its grade: 3.0, 4.0/4.5, 5.0.
- LAG 0 = the worker has processed every message.
- These are real outputs from a local run, not mock-ups.

Synchronous path — what happens when you press "Play now"

SYNCHRONOUS PATH

PART 2 · GLORIA

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;
}
```

Mezarpoly · Docker & Azure

12

WHAT TO SAY

Let us follow one click. When you open the site, nginx serves the HTML, JavaScript and CSS from its own disk — the backend is not involved.

Then the client opens a WebSocket at `/ws`. A reverse proxy normally closes a connection after each request, so nginx must forward the **Upgrade** and **Connection** headers; otherwise the WebSocket never opens.

We also raised the read timeout to one hour. The default is sixty seconds, which would disconnect a player who thinks for a minute.

The backend then creates the room, the bots and the board in memory and starts the game. Accounts, results and the shop are read and written in PostgreSQL. All of this is synchronous: the browser waits for the answer.

REMEMBER

- WebSocket through a proxy needs the Upgrade + Connection headers and HTTP/1.1.
- `proxy_read_timeout 3600s`.

Asynchronous path — Kafka and the worker

ASYNCHRONOUS PATH
PART 2 · GLORIA

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"}
  
```

Mezarpoly · Docker & Azure 13

WHAT TO SAY

This is the part for grade 5.0. When a match starts or ends, the backend acts as a **producer**: it puts a small JSON event on the Kafka topic `mezarpoly.events` and continues immediately. We call this "fire and forget". If Kafka is slow or even down, the game does not wait.

The **worker** is the **consumer**, in the consumer group `mezarpoly-stats`. For each event it writes a row to the `game_events` table and increases a counter in `event_counts`, both in one database transaction.

The important detail is the order: it writes to the database **first** and only then **commits the offset** — tells Kafka "I have processed this message". If the worker crashed in between, Kafka would deliver the same message again, so no event is ever lost. The price is that, rarely, an event could 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.

REMEMBER

- Producer = backend; consumer = worker; topic = `mezarpoly.events`; group = `mezarpoly-stats`.
- Commit after write → at-least-once.
- Fire and forget → the game never blocks on Kafka.

Database — one data layer, two databases

DATABASE

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>

Mezarpoly · Docker & Azure

14

WHAT TO SAY

In our five-container design the database lives in its own container, so it is a database server reachable over the network: PostgreSQL.

In the light single-server mode the public site uses SQLite, an embedded single-file database. So from the start we needed one data layer that works with both.

We wrote the data layer once, against Python's small `sqlite3` interface, and added an **adapter** of about 150 lines. It offers the same interface but talks to PostgreSQL and translates each query: question-mark placeholders become `%s`, `INSERT OR IGNORE` becomes `ON CONFLICT DO NOTHING`, `AUTOINCREMENT` becomes `BIGSERIAL`, and so on.

In the container setup the variable `MP_DATABASE_URL` selects PostgreSQL; without it the code uses SQLite. And the proof: our automated test suite passes against PostgreSQL too.

REMEMBER

- One code base for both modes: the adapter, not two versions of the SQL.
- Proof = our test suite runs green on PostgreSQL.

Why Azure Container Apps

CLOUD PLATFORM

PART 2 · GLORIA

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

Mezarpoly · Docker & Azure

15

WHAT TO SAY

For the cloud we compared four Azure options.

A **virtual machine** running Docker Compose would work, but it is one machine — not separate components — and we would have to maintain the operating system.

Azure Kubernetes Service is the industry standard, but it is heavy to learn and to run for a project this size.

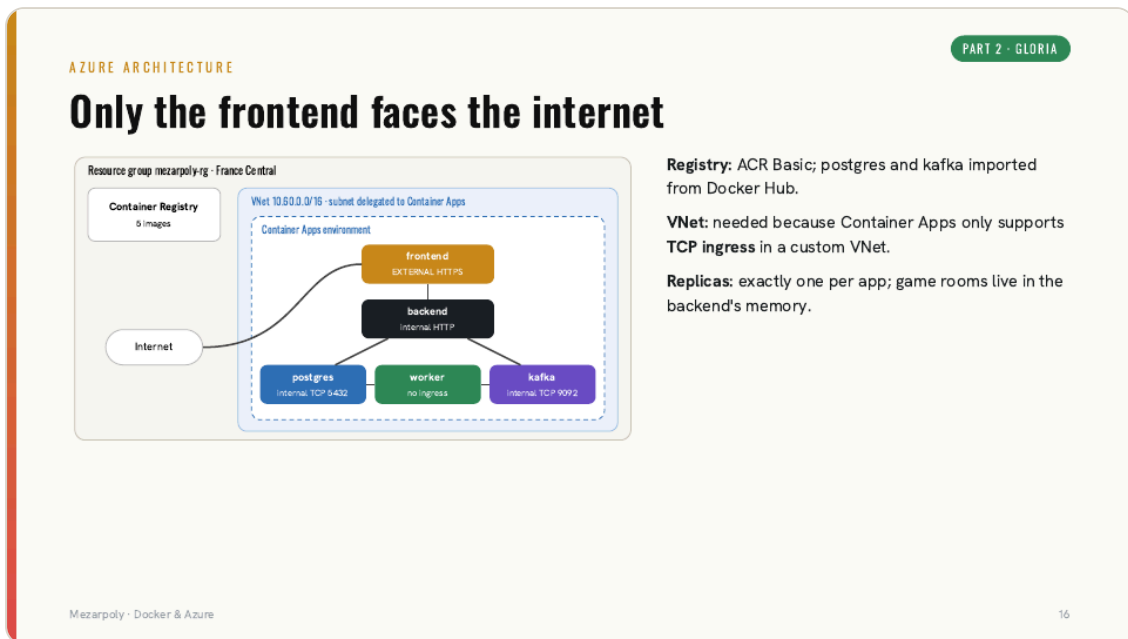
App Service is excellent for web applications, but it cannot run TCP services such as PostgreSQL and Kafka as containers.

Container Apps gives every container its own app, internal and external ingress, TCP support and pay-per-use pricing. It runs on Kubernetes, but Microsoft manages it for us. That is why we chose it.

REMEMBER

- One sentence per option; the decisive reasons: separate components + TCP + managed.

Azure architecture — only the frontend faces the internet



WHAT TO SAY

This is the cloud version of Azim's architecture. Everything lives in one **resource group** in France Central, one of the regions our student subscription allows.

The five images are in an **Azure Container Registry**.

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 placed the environment inside it.

The ingress types mirror the Docker networks: only the **frontend** is external, with HTTPS and a certificate from Azure. The backend is internal HTTP, PostgreSQL and Kafka are internal TCP, and the worker has no ingress at all, because it only makes outgoing connections.

Each app runs exactly one replica: game rooms live in the backend's memory, so two backends could not share a room.

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."

REMEMBER

- External = frontend only; internal HTTP = backend; internal TCP = postgres, kafka; none = worker.
- VNet is needed for TCP ingress.

Background you should understand

Docker Compose in a few lines

```
services:
  backend:
    depends_on:
      postgres: { condition: service_healthy }
      kafka: { condition: service_healthy }
    networks: [web, data]
  frontend:
    depends_on: { backend: { condition: service_healthy } }
    ports: ["8080:80"] # host port 8080 → container port 80
    networks: [web]
networks: { web: {}, data: {} }
```

`ports` publishes a container port on the host computer: only the frontend has one, so only it is reachable from outside. Volumes (`pg-data`, `kafka-data`) keep data when containers are recreated; `docker compose down -v` deletes them.

Reverse proxy and WebSocket

A **reverse proxy** receives every request and decides where it goes. nginx serves files from disk for `/`, and passes `/api`, `/auth` and `/ws` to the backend. A WebSocket begins as an HTTP request with `Upgrade: websocket`; the proxy must pass those headers and use HTTP/1.1 so the connection can be "upgraded" to a permanent two-way channel. nginx also re-resolves the backend's name on each request (`resolver` + variable in `proxy_pass`), so a restarted backend with a new IP is still found.

Kafka concepts

Term	Meaning	In our project
Broker	Kafka server that stores messages	the kafka container
Topic	Named channel of messages	mezarpoly.events
Partition	Ordered sub-log of a topic; unit of parallelism	1
Producer	Writes messages	backend (events.py)
Consumer	Reads messages	worker (worker.py)
Consumer group	Consumers sharing the work; Kafka remembers its position	mezarpoly-stats
Offset	Position of a message in a partition	committed after the DB write
KRaft	Kafka without ZooKeeper	single node

Delivery guarantees

- **At-most-once:** commit before processing — a crash can lose a message.
- **At-least-once** (ours): commit after processing — no loss, but a message can be processed twice.
- **Exactly-once:** needs transactions across Kafka and the database or idempotent writes; more complex, not needed here.

Why it is acceptable to store an event twice here: these are statistics, not money. To make duplicates impossible we could add a unique key (topic + partition + offset) to `game_events`.

The worker's loop

```
async for msg in consumer:
    async with db.transaction():          # both writes or none
        INSERT INTO game_events(type, ts, data) ...
        INSERT INTO event_counts(type, n) VALUES (%s, 1)
        ON CONFLICT (type) DO UPDATE SET n = event_counts.n + 1
    await consumer.commit()              # only after the write
```

The worker and the backend both **retry** the Kafka connection (every 2 s, 30–60 times) because Kafka may become ready after them. A harmless `GroupCoordinatorNotAvailableError` can appear in the logs while Kafka creates its internal topics on first start.

The PostgreSQL adapter

SQLite	PostgreSQL
?	%s
INSERT OR IGNORE	INSERT ... ON CONFLICT DO NOTHING
INTEGER PRIMARY KEY AUTOINCREMENT	BIGSERIAL PRIMARY KEY
MAX(a, b) / MIN(a, b)	GREATEST / LEAST
ADD COLUMN	ADD COLUMN IF NOT EXISTS
cursor.lastrowid	INSERT ... RETURNING id
PRAGMA ...	ignored

Azure building blocks you will mention

- **Resource group** — a folder for related resources; deleting it deletes everything inside.
- **Region** — the data-centre location (France Central).
- **Container Registry (ACR)** — private image store; postgres and kafka were imported from Docker Hub.
- **VNet + delegated subnet** — a private network; the subnet is handed to Container Apps (minimum size /23).

- **Container Apps environment** — the secure boundary where the apps live and find each other by name.
- **Ingress** — how an app is reached: external HTTPS, internal HTTP, internal TCP, or none.
- **Replica** — a running instance; we use exactly one per app.

Questions you may get — with model answers

Read each question, answer it out loud in your own words, then compare with the model answer.

1. What is the difference between `depends_on` and `condition: service_healthy`?

`depends_on` only controls the start order; it does not wait for the service to be ready. `service_healthy` makes Compose wait until the container's health check passes.

2. How does the backend know where PostgreSQL is?

Through the service name: it connects to `postgres:5432` and Docker's internal DNS resolves the name on the shared network. In Azure the same name works inside the Container Apps environment.

3. Why does nginx need special settings for WebSocket?

A WebSocket starts as an HTTP request with Upgrade headers. nginx must forward the Upgrade and Connection headers and use HTTP/1.1; otherwise the connection is not upgraded. We also raised `proxy_read_timeout` so idle games are not cut after 60 seconds.

4. Why use Kafka at all — the backend could write statistics directly?

Writing directly would make the game wait for the database for something that is not urgent, and a slow database would slow the game. With Kafka the game publishes and continues; the worker processes events at its own pace, and if the worker is down, events wait in Kafka.

5. What happens if Kafka is down?

The game continues normally. Publishing is fire-and-forget; a failed send is only logged.

6. What happens if the worker crashes?

Events stay in Kafka. When the worker restarts, it continues from the last committed offset, so nothing is lost.

7. What is at-least-once delivery and its downside?

The offset is committed only after processing, so a message is never lost; the downside is that a message can be processed twice after a crash.

8. How could you avoid duplicates?

Make the write idempotent, e.g. store `topic+partition+offset` with a unique constraint and ignore conflicts.

9. Why not write separate SQL for PostgreSQL?

Two copies of every query would drift apart and double the bugs. One data layer plus a small adapter keeps a single code base; the test suite proves it works on both databases.

10. Why not a virtual machine with Docker Compose?

It is one machine, so the containers would not be separate cloud components, and we would have to patch and secure the operating system ourselves.

11. Why was a VNet necessary?

Container Apps supports TCP ingress — needed by PostgreSQL and Kafka — only in environments deployed into a custom virtual network.

12. Why does the worker have no ingress?

Nobody needs to connect to it; it only makes outgoing connections to Kafka and PostgreSQL. No ingress means no attack surface.

Terms in your part

Docker Compose

Tool that runs a multi-container application from one YAML file.

depends_on

Compose option that orders service start-up.

Volume

Persistent storage for container data.

Reverse proxy

Front door that forwards requests to internal servers.

Upgrade header

HTTP header that turns a request into a WebSocket.

Producer / consumer

Program that writes / reads Kafka messages.

Topic

Named message channel in Kafka.

Offset

Position of a message; committed when processed.

Consumer group

Consumers that share a topic's work.

At-least-once

No message is lost; duplicates are possible.

Transaction

Group of database writes that succeed or fail together.

Ingress

Rule for how an app is reached (external, internal, TCP).

VNet

Private virtual network in Azure.

Resource group

Azure folder for related resources.

Container Registry

Private store for container images.

Checklist before you present

- ☐ Open Slides-Part2-Gloria.pdf full screen and test screen sharing before the presentation; turn on do not disturb.
- ☐ Rehearse your part out loud twice with a timer (target: about 4 minutes).
- ☐ Be able to explain at-least-once in two sentences without the slide.
- ☐ Know the topic name, the consumer group and the two tables the worker writes.
- ☐ Know the four ingress types and which app has which.
- ☐ Practise the opening thank-you and the handover sentence to Kutay.
- ☐ Agree who holds the clicker during your part.