

# 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

## DOCKER COMPOSE

# 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`.

# 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.

## SYNCHRONOUS PATH

# 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;  
}
```

## ASYNCHRONOUS PATH

# 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"}
```

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.

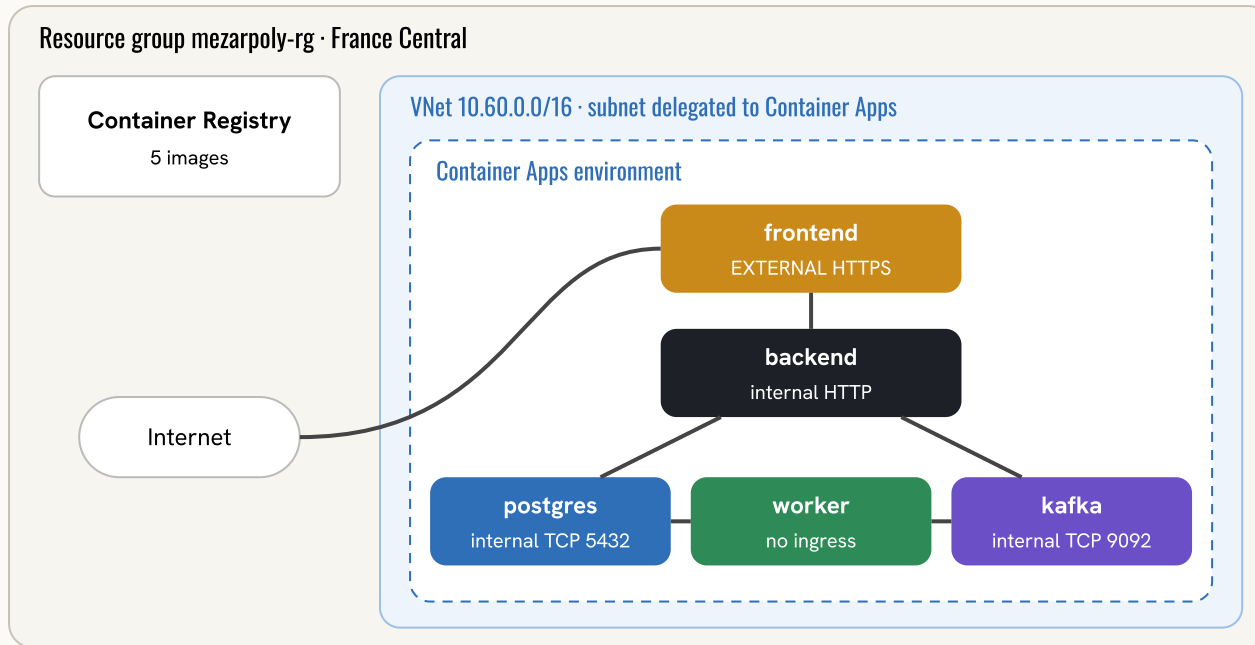
In the container setup `MP_DATABASE_URL` selects PostgreSQL; the light single-server mode uses SQLite. Our whole test suite runs against PostgreSQL to prove it.

| 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>           |

# 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                                                     |
| <b>Container Apps</b>            | <b>Chosen:</b> one app per container, internal/external ingress, TCP support, pay per use, Kubernetes managed by Azure |

# 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.