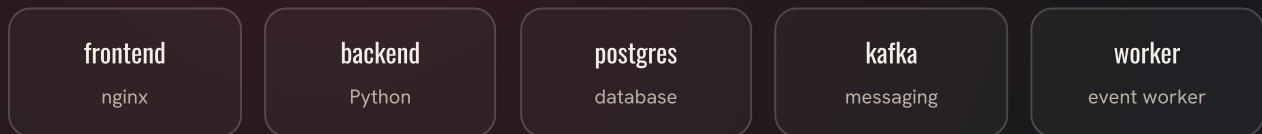


Moving Mezarpoly to the cloud with Docker and Azure

How a real multiplayer game made of five containers runs on a laptop with Docker Compose and then as separate cloud components on Microsoft Azure Container Apps — explained step by step, assuming no prior knowledge.



Contents

- 0 Who this guide is for and how to read it
- 1 The project: Mezarpoly and the assignment
- 2 Basic concepts from scratch: server, client, port, HTTP, WebSocket
- 3 What is a container, and how does Docker work?
- 4 The project's Dockerfiles, line by line
- 5 Docker Compose: running five containers together
- 6 Architecture: the journey of one click
- 7 nginx: the front door and reverse proxy
- 8 PostgreSQL and the shared data layer
- 9 Apache Kafka and asynchronous communication
- 10 Introduction to cloud computing and Microsoft Azure
- 11 The building blocks of Azure Container Apps
- 12 deploy.sh: the Azure deployment step by step
- 13 Problems we hit and how we solved them
- 14 Verification: proof that it works
- 15 Cost, stopping and restarting
- 16 Live demo script for the presentation
- 17 Command cheat sheet
- 18 Glossary
- 19 Test yourself: 25 questions and answers

Presentation team

Part 1 — **Azim**: the application, architecture, containers and images (chapters 1–5). Part 2 — **Gloria**: Compose, the communication paths, the database and the cloud platform (chapters 5–11). Part 3 — **Kutay**: deployment, security, troubleshooting and proof (chapters 12–16). Each presenter also has a separate presenter guide.

0 - Who this guide is for and how to read it

This guide is written so that someone who has never heard of "Docker" or "Azure" can, by the end, answer these questions in their own words: Why is an application split into parts? How do the parts talk to each other? How do I move a system that runs on my computer into the cloud? What goes wrong on the way, and how is it fixed?

How to read it

- **If you know nothing yet:** read chapters 2 and 3 without skipping. Each new concept is first explained with an everyday analogy, then defined technically.
- **If you know Docker:** start at chapter 5 (Compose) and go on to the architecture in chapter 6; Azure starts in chapter 10.
- **If you are preparing the presentation:** chapters 6, 13, 14 and 16 and the questions at the end are the most useful.

What the boxes mean

Analogy

Explains a new idea with an everyday example.

Info

Extra explanation that goes deeper.

Tip

A shortcut or habit that helps in practice.

Watch out

A common mistake, or a problem that really happened in this project.

Reading code blocks

Lines in the dark boxes are commands typed in a terminal or file contents. Grey lines starting with `#` are comments; the computer ignores them.

```
# This is a comment
docker compose up --build    # this line is a command
```

1 - The project: Mezarpoly and the assignment

Mezarpoly is a real-time, browser-based property-trading game for 2-8 players. Players roll dice, buy cities, build houses and hotels, trade and bid in auctions. We — Azim, Gloria and Kutay — built it, and it runs publicly at mezarpoly.fun.

The story of the project

We designed Mezarpoly from the start as a system of **five services**: a web front (nginx), the game server, a database, a message broker (Kafka) and a worker. Every part is switched on by configuration. When we put the public site online, we saw 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 we planned it, locally and on Microsoft Azure.



Figure 1.1 — The Mezarpoly game screen (captured from the game's Azure deployment).

The goal of the assignment

The project for the course "Cloud-oriented web applications" asks us to build the application from **containers**, run them together locally, and then deploy each of them as a separate component in the cloud — which is exactly the architecture we designed Mezarpoly for. The grade rises step by step with the features delivered:

Grade	Requirement	How this project meets it
3.0	The application starts in Docker with <code>docker-compose.yml</code>	One command (<code>docker compose up --build</code>) starts all five containers.
4.0	Two containers communicate over a Docker network (frontend ↔ backend)	The frontend (nginx) forwards HTTP and WebSocket requests to the backend over the <code>web</code> network.
4.5	A third container: a database; the backend talks to the database and the frontend	postgres on the <code>data</code> network stores accounts, game results, the shop and friends.
5.0	A fourth container for asynchronous communication (e.g. Kafka)	kafka + worker : the backend writes game events to Kafka; the worker reads them and stores them in PostgreSQL.
6.0	All containers run in the cloud as separate components	Microsoft Azure Container Apps: each container is its own Container App, created by one script: <code>docker/azure/deploy.sh</code> .

Why so many parts?

Large real systems are split into small parts that each do one job well, instead of one giant program. If one part fails, the others can keep running; each part can be scaled separately; and different people can work on different parts. Here, each of the five parts has one job.

Five parts, five jobs

Container	Its job in one sentence	Software
frontend	Gives the browser the game's files and routes requests to the right place.	nginx 1.29
backend	The brain of the game: rules, rooms, bots, accounts, live connections.	Python 3.13 + aiohttp
postgres	Stores permanent data (accounts, results...).	PostgreSQL 17
kafka	A mailbox for events: stores messages in order and hands them out.	Apache Kafka 3.9 (KRaft)
worker	Reads the events from the mailbox and writes them to the database.	Python 3.13 + aiokafka

2 · Basic concepts from scratch

To understand Docker and Azure you first need the basic building blocks of a web application. Skip this chapter if you know them.

2.1 Client and server

Analogy: a restaurant

You (the customer) order from the waiter; the kitchen cooks and the waiter brings the food. On the web your **browser** is the customer (the client) and the remote computer is the kitchen (the server). The client sends a request; the server sends a response.

A **server** is a program that is always running and answers incoming requests (the word is also used for the computer it runs on). A **client** is the side that starts the request: a browser, a mobile app, or even another server.

2.2 IP address and port

Analogy: an apartment building

An IP address is like the street address of a building; a **port** is the flat number. Many programs can run on one computer, each listening on a different "flat" (port). In Mezarpoly nginx listens on port 80, the game server on 8095, PostgreSQL on 5432 and Kafka on 9092.

Example: `localhost:8080` means "port 8080 on this computer". `localhost` always means "my own computer".

2.3 HTTP: request and response

The browser and the server speak a common language called **HTTP**. Every exchange is a **request** and a **response**. A request has a **method** (GET = "give me", POST = "take/save this") and a **path** (like `/api/leaderboard`). A response has a **status code** and content.

Status code	Meaning	Where we saw it
200	OK, here it is	A successful health check
301 / 302	Go to another address	Redirects
403	Forbidden	POST requests from an unknown origin (CSRF protection)
404	Not found	Calling the backend by its short name in Azure (chapter 13)
502	The server behind could not be reached	nginx's answer while the backend was not up yet

HTTPS is the encrypted form of HTTP: nobody in between can read the conversation. Azure provides the encryption certificate for the public app automatically.

2.4 WebSocket: a line that stays open

Analogy: letters and phone calls

HTTP is like writing letters: every message is a new letter. A **WebSocket** is like a phone call: the line is opened once and either side can talk at any moment. When one player rolls the dice and every other screen updates *immediately*, it is because the server can tell them without being asked.

In Mezarpoly the game itself runs over a WebSocket at `/ws`. A WebSocket starts as an HTTP request and turns into a permanent line using the *Upgrade* header. This detail comes back in chapter 7 with the nginx settings.

2.5 Database

When a program stops, everything in memory is lost. Data that must last (accounts, password hashes, games won, items bought) is stored in a **database**. **Relational databases** (PostgreSQL, SQLite, MySQL) keep data in **tables** of rows and columns and are queried with a language called **SQL**:

```
SELECT name, points FROM users ORDER BY points DESC LIMIT 10;
-- "give me the names and points of the 10 users with the most points"
```

2.6 Synchronous and asynchronous communication

Synchronous

You phone someone and wait for the answer; you cannot do anything else until they reply. Example: pressing "log in" and waiting for the server to say "password correct".

Asynchronous

You leave a note and carry on; the note is read when its turn comes. Example: leaving a "game finished" event in Kafka; the worker processes the statistics later.

Grade 5.0 of this project is exactly about this difference: game events travel on an asynchronous path (Kafka), so the game never waits for statistics to be saved.

3 - What is a container, and how does Docker work?

3.1 The problem: "It worked on my machine"

When you move a program to another computer, it often does not run: the Python version is different, a library is missing, a configuration file is absent. Developers even have a name for it: *"works on my machine"*.

Analogy: shipping containers

Cargo used to be loaded onto ships sack by sack, differently at every port. Then standard steel **containers** were invented: whatever is inside, every crane, ship and truck handles them the same way. Software containers are the same idea: the program is packed with everything it needs into a standard box, and that box runs the same way on any computer with Docker.

3.2 Container vs. virtual machine

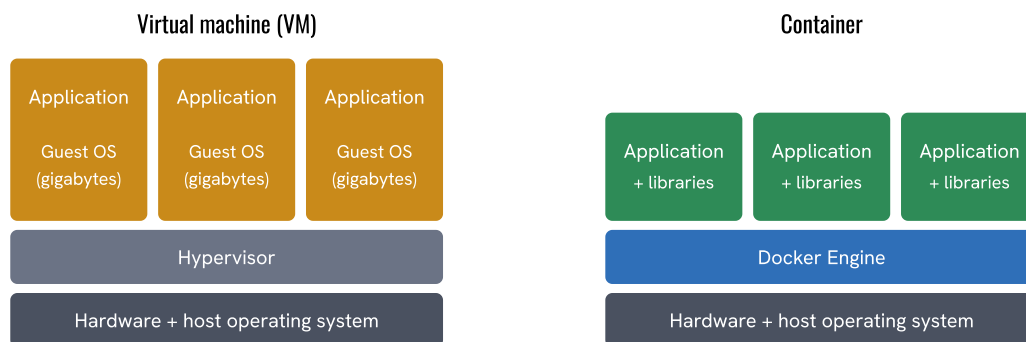


Figure 3.1 — Every virtual machine carries its own operating system; containers share the host's kernel, which makes them much lighter and faster.

A **virtual machine** imitates a whole computer inside your computer: its own operating system and memory; it takes minutes to boot and gigabytes of space. A **container** shares the host's operating-system kernel and only carries the files the application needs; it starts in seconds and takes megabytes.

3.3 Image, container, registry

Concept	Analogy	Definition
Image	A recipe / a mould	A frozen, read-only package of the application and everything it needs, e.g. <code>postgres:17-alpine</code> .
Container	The dish cooked from the recipe	A running copy of an image. Many containers can start from one image.
Dockerfile	The written recipe	A text file describing step by step how to build an image.
Registry	A recipe library	Where images are stored and downloaded: Docker Hub, Azure Container Registry.
Tag	The edition number	The version of an image: the <code>v2</code> in <code>mezarpoly-backend:v2</code> .
Volume	A fridge	Persistent storage that keeps data even if the container is deleted.

3.4 Layers and the cache

An image is made of **layers** stacked on each other; every Dockerfile instruction produces one. If Docker built a layer before and nothing affecting it changed, it reuses it from the cache. That is why Dockerfiles copy rarely-changing things (the library list) before frequently-changing things (the source code).

Tip

The backend Dockerfile copies `requirements.txt` and installs the libraries first, and copies the source code afterwards. When one line of code changes, the libraries are not downloaded again and the build takes seconds.

3.5 Docker networks

Containers talk over virtual networks created by Docker. Containers on the same network find each other by **service name**: the backend reaches the database as `postgres:5432` without knowing its IP address, because Docker runs its own name resolver (DNS). **Containers on different networks cannot see each other** — an important security feature (chapter 5).

3.6 Health checks

A container being "running" does not mean the program inside is ready: PostgreSQL refuses connections for a few seconds after it starts. A **health check** is a small test Docker runs at intervals. If it passes, the container is *healthy*. Compose can wait for this state before starting other containers.

4 · The project's Dockerfiles, line by line

Three custom images are built (frontend, backend, worker). PostgreSQL and Kafka use official images.

4.1 backend — the game server

```
FROM python:3.13-slim # 1
WORKDIR /app # 2
COPY requirements.txt docker/backend/requirements-docker.txt ./ # 3
RUN pip install --no-cache-dir -r requirements.txt -r requirements-docker.txt \
    && useradd -r -u 10001 app && mkdir -p /data && chown app /data # 4
COPY server/ ./server/ # 5
WORKDIR /app/server
USER app # 6
ENV MP_HOST=0.0.0.0 MP_PORT=8095 MP_DB=/data/accounts.db MP_STATE=/data/state.json PYTHONUNBUFFERED=1 # 7
EXPOSE 8095 # 8
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)" # 9
CMD ["python", "server.py"] # 10
```

1. **FROM** — The base image: a small Debian with Python 3.13, stripped of unneeded packages ("slim").
2. **WORKDIR** — The working folder inside the container; later commands run here.
3. **COPY** — Only the library lists are copied first (for the layer cache, see 3.4).
4. **RUN** — Libraries are installed; an unprivileged user `app` and a data folder are created.
5. **COPY** — The game server's source code is copied.
6. **USER** — The program runs as a normal user, not `root`. Even if a flaw is found, an attacker's rights inside the container stay limited.
7. **ENV** — Environment variables: the program reads its settings from them. `0.0.0.0` means "listen on all network interfaces"; needed for access from outside the container.
8. **EXPOSE** — Documentation: "this container listens on 8095".
9. **HEALTHCHECK** — Every 10 seconds `/healthz` is called; if it answers, the container is healthy.
10. **CMD** — The command run when the container starts.

4.2 frontend — a multi-stage build

The frontend has two jobs: **building** the game's browser files and **serving** them. Building needs Node.js; serving does not. With a **multi-stage build**, the first stage builds, and only the result is copied into the second stage. The final image contains no Node.js or build tools: it is small and safe.

```
# Stage 1 (build): compile the Vue templates and assemble the static site
FROM node:22-alpine AS build
RUN apk add --no-cache bash python3
WORKDIR /src
COPY tools/package.json tools/package-lock.json* tools/
RUN cd tools && npm install --omit-dev --ignore-scripts --no-audit --no-fund @vue/compiler-dom@^3.4.21
COPY tools/build.js tools/dist.sh tools/
COPY client/ client/
RUN bash tools/dist.sh

# Stage 2 (runtime): nginx serves the files and proxies /api /auth /ws to the backend
FROM nginx:1.29-alpine
ENV BACKEND_URL=http://backend:8095 NGINX_ENTRYPOINT_LOCAL_RESOLVERS=1
COPY docker/frontend/default.conf.template /etc/nginx/templates/default.conf.template
COPY --from=build /src/dist/ /usr/share/nginx/html/
COPY docker/frontend/status.html /usr/share/nginx/html/status.html
EXPOSE 80
HEALTHCHECK --interval=10s --timeout=3s --retries=5 CMD wget -q -O /dev/null http://127.0.0.1/status.html
|| exit 1
```

Why is BACKEND_URL an environment variable?

The nginx configuration file is a **template**: `${BACKEND_URL}` is filled in from the environment when the container starts. Locally the value is `http://backend:8095`; on Azure it is `http://backend.internal.<environment domain>`. The same image runs in both places with different settings. This principle is called **separating configuration from code**.

4.3 worker — the event worker

```
FROM python:3.13-slim
WORKDIR /app
RUN pip install --no-cache-dir "aiokafka≥0.11" "psycpg[binary]≥3.2" && useradd -r -u 10001 app
COPY docker/worker/worker.py .
USER app
CMD ["python", "-u", "worker.py"]
```

The worker's job is small, so its image is small: two libraries (the Kafka client and the PostgreSQL driver) and one Python file. `-u` makes the output go to the logs immediately, without buffering.

5 · Docker Compose: running five containers together

Analogy: a conductor

If each container is a musician, **Docker Compose** is the conductor. One score (`docker-compose.yml`) says who plays what, who comes in after whom and who plays with whom; one gesture (`docker compose up`) starts everyone.

5.1 The important parts of docker-compose.yml

```
services:
  frontend:
    build: { context: ., dockerfile: docker/frontend/Dockerfile }
    ports: ["8080:80"] # computer's 8080 → container's 80
    environment:
      BACKEND_URL: http://backend:8095 # the backend is reached by service name
    depends_on:
      backend: { condition: service_healthy } # do not start until the backend is healthy
    networks: [web]

  backend:
    build: { context: ., dockerfile: docker/backend/Dockerfile }
    environment:
      MP_DATABASE_URL: postgres://mezarpoly:${POSTGRES_PASSWORD:-mezarpoly-dev}@postgres:5432/mezarpoly
      MP_KAFKA: kafka:9092
    depends_on:
      postgres: { condition: service_healthy }
      kafka: { condition: service_healthy }
    networks: [web, data] # on both networks: the bridge

  postgres:
    image: postgres:17-alpine
    volumes: ["pg-data:/var/lib/postgresql/data"] # persistent data
    healthcheck: { test: ["CMD-SHELL", "pg_isready -U mezarpoly -d mezarpoly"], interval: 5s, retries: 20 }
    networks: [data]

  kafka:
    image: apache/kafka:3.9.1
    environment: { KAFKA_PROCESS_ROLES: "broker,controller", ... }
    networks: [data]

  worker:
    build: { context: ., dockerfile: docker/worker/Dockerfile }
    networks: [data]

networks:
  web: {} # frontend ↔ backend
  data: {} # backend ↔ postgres / kafka / worker
```

5.2 Two networks: why?

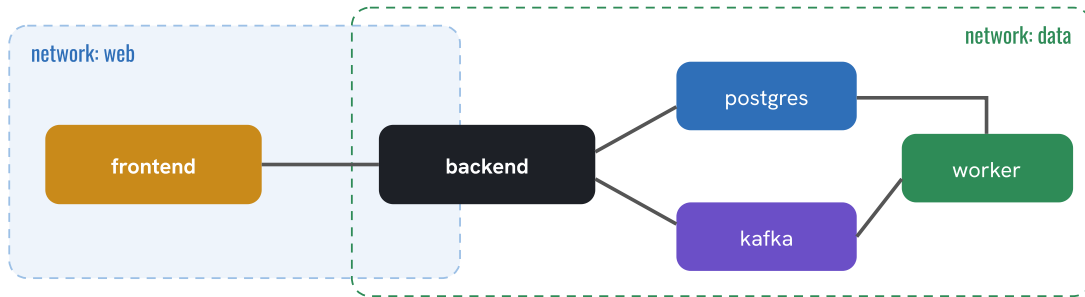


Figure 5.1 — The frontend is only on **web**; the database and Kafka are only on **data**. The only container connecting them is the backend.

The frontend is the only container exposed to the internet. Even if an attacker found a flaw in nginx, the **web** network it lives on contains no database; there is no direct route to PostgreSQL. This is called **network segmentation** — the network form of the "least privilege" principle.

5.3 Start-up order

Thanks to `depends_on` and `condition: service_healthy`, Compose follows this order:

- 1 **postgres** and **kafka** start. Compose waits until their health checks pass (`pg_isready` for PostgreSQL; listing topics for Kafka).
- 2 **backend** and **worker** start. The backend is healthy when `/healthz` answers.
- 3 **frontend** starts and opens to the outside world.

5.4 Running it

```
docker compose up --build -d      # build 3 images, pull 2, start everything in the background
docker compose ps                 # 5 services "Up", 4 of them "(healthy)"
# Browser: http://localhost:8080 - "Play now"
# Live status: http://localhost:8080/status.html
docker compose logs -f worker     # watch the worker process events live
docker compose down               # stop (data stays in the volumes)
docker compose down -v            # stop and delete all data
```

Tip: seeing the networks

```
docker network inspect mezarpoly_web --format '{{range .Containers}}{{.Name}} {{end}}'
docker network inspect mezarpoly_data --format '{{range .Containers}}{{.Name}} {{end}}'
```

The first command lists only frontend and backend, the second backend, postgres, kafka and worker: proof of the network separation.

6 · Architecture: the journey of one click

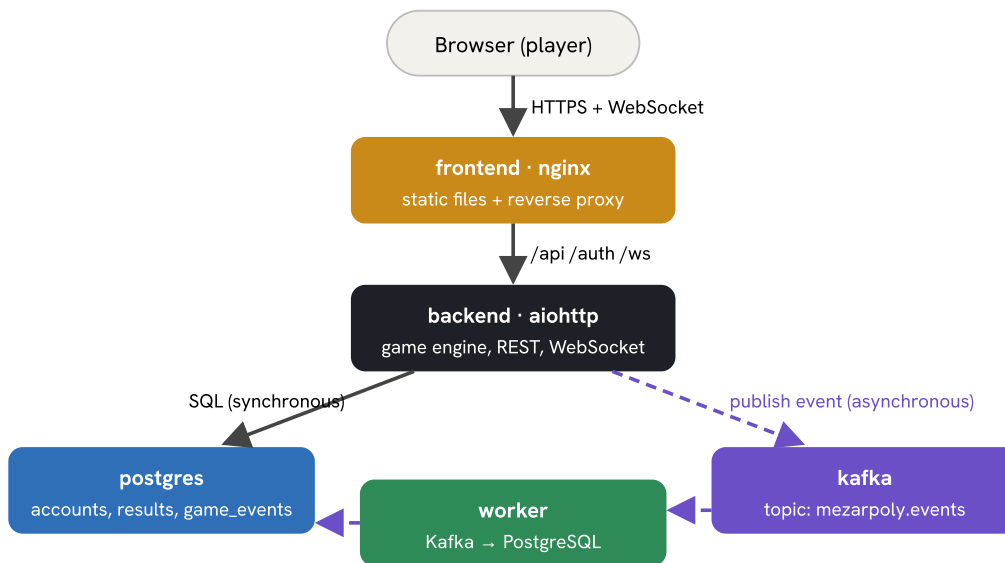


Figure 6.1 — Solid arrows: the synchronous path (an answer is awaited). Dashed purple arrows: the asynchronous event path.

6.1 What happens when you press "Play now"?

- 1 The page loads.** The browser requests `/`. nginx returns the compiled HTML, JavaScript and CSS from its own disk; the backend is not involved.
- 2 The live line opens.** The game opens a WebSocket to `/ws`. nginx forwards it to the backend together with the "upgrade" headers (chapter 7). Now there is a permanent line between the browser and the backend, passing through nginx.
- 3 The room is created and the game starts.** The browser sends `{"t": "create"}` over the line. The backend builds the room, the bots and the board in memory, starts the game and sends every player the initial state.
- 4 An event is published (asynchronously).** The backend leaves a `game_started` event in Kafka and continues **without waiting**. Even if Kafka is slow or down at that moment, the game is not affected.
- 5 The worker processes the event.** The worker takes the event from Kafka, writes it to the `game_events` table in PostgreSQL and increments the `event_counts` counter. Then it tells Kafka "I have processed this" (offset commit, chapter 9).
- 6 The result comes back.** The `/status.html` page regularly asks `/api/events/summary`; the backend reads that from PostgreSQL and returns it. The data has made a full circle: **frontend** → **backend** → **Kafka** → **worker** → **database** → **backend** → **frontend**.

6.2 An example event

```
{ "type": "game_finished", "ts": 1791818816, "room": "PHAEJ", "board": "classic",  
  "turns": 1, "humans": 2, "winner": "DockerB", "minutes": 0.1 }
```

An event is a small message in plain text (JSON): what happened (`type`), when (`ts` , a Unix timestamp), in which room and with which details.

6.3 One code base, two modes

We built the game so that every container-related part is switched on by **environment variables**. The same code runs as the full five-container system or, when the variables are not set, in a light single-server mode. The public site `mezarpoly.fun` uses the light mode, because at its current size the extra containers were not needed:

Variable	If set (five containers)	If not set (light mode)
<code>MP_DATABASE_URL</code>	PostgreSQL is used (<code>server/pgcompat.py</code>)	An SQLite file is used
<code>MP_KAFKA</code>	Events are published to Kafka (<code>server/events.py</code>)	Event publishing is off

7 · nginx: the front door and reverse proxy

Analogy: a hotel reception

A hotel reception greets everyone. It hands out brochures itself and connects room-service requests to the right department; visitors never see how many departments there are behind it. nginx does the same: it serves static files (the brochures) itself and forwards `/api`, `/auth` and `/ws` to the backend. This is called a **reverse proxy**.

7.1 The important parts of the configuration

```
server {
    listen 80;
    root /usr/share/nginx/html;
    resolver ${NGINX_LOCAL_RESOLVERS} valid=10s ipv6=off; # resolve names on every request
    set $backend ${BACKEND_URL}; # address from the template

    location ~ ^/(api|auth)/ { # REST requests
        proxy_pass $backend;
        proxy_set_header Host $proxy_host;
        proxy_set_header CF-Connecting-IP $client_ip; # the real client IP
    }
    location = /ws { # the live game line
        proxy_pass $backend;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade; # upgrade to WebSocket
        proxy_set_header Connection $connection_upgrade;
        proxy_read_timeout 3600s; # the line may be quiet for 1 hour
    }
    location / { try_files $uri $uri.html $uri/ /index.html; } # static files
}
```

7.2 Three critical details

- **The WebSocket upgrade:** by default nginx closes the connection after each request. If the `Upgrade` and `Connection` headers are not passed on to the backend, the game line cannot be established.
- **proxy_read_timeout:** the default is 60 seconds; if a player does not move for a minute, nginx cuts the line. So it was raised to one hour.
- **resolver + a variable in proxy_pass:** nginx normally resolves addresses only at start-up. If the backend starts later or its IP changes on Azure, nginx would keep going to the old address. Reading the address from a variable and defining a resolver makes nginx resolve the name on every request.

Watch out: the real IP

Behind a reverse proxy, the backend sees nginx's IP, not the user's. The game's rate limits and anti-cheat rules work per IP, so nginx passes the user's real IP in a separate header. On Azure there is one more layer in front (Azure's load balancer), which writes the real IP into `X-Forwarded-For`; nginx takes its first value.

8 · PostgreSQL and the shared data layer

8.1 Why PostgreSQL?

In the five-container design the database lives in its own **separate container**, which means a database server reached over the network: **PostgreSQL**. In the light single-server mode an embedded **SQLite** file is enough — it keeps everything in one file and needs no separate server.

8.2 One data layer for two SQL dialects

SQL is a standard, but every database has small differences. We wrote the game's data layer once, against Python's small `sqlite3` interface, and added a thin **adapter** (`server/pgcompat.py` , about 150 lines) that offers the same interface for PostgreSQL and translates each query before sending it:

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

How was it verified?

Our automated test suite runs with PostgreSQL as the database as well as with SQLite. This proves the adapter works with the real code without writing a separate test for every query.

8.3 Where is the data kept?

Locally, PostgreSQL's data lives in a Docker volume called `pg-data` ; it survives `docker compose down` . On Azure no persistent disk was attached in this project: if the postgres app restarts, the database starts empty. That is an acceptable simplification for the assignment; a real system would use Azure's managed PostgreSQL service (Azure Database for PostgreSQL) — see chapter 15.

9 - Apache Kafka and asynchronous communication

Analogy: a notice board

Think of a board in a school corridor. Teachers (producers) pin announcements and go back to class; they do not wait to see who reads them. Students (consumers) look at the board whenever they like, and each remembers where they stopped reading. Announcements stay for a while, so someone who comes later can read older ones too. Kafka is that board.

9.1 Core concepts

Concept	Meaning	In this project
Broker	The Kafka server that stores messages	the <code>kafka</code> container (single node)
Topic	A named channel that messages are written to	<code>mezarpoly.events</code>
Partition	An ordered sub-part of a topic; gives parallelism	1 partition
Producer	A program that writes messages to a topic	backend (<code>events.py</code>)
Consumer	A program that reads messages from a topic	worker (<code>worker.py</code>)
Consumer group	Consumers that share the work	<code>mezarpoly-stats</code>
Offset	A message's position in a partition; "where I stopped"	committed by the worker after processing
KRaft	Kafka's mode without ZooKeeper	broker and controller in one container

9.2 The producer: the game never waits

```
def emit(self, kind, **data):
    if not self.producer:
        return # no Kafka → skip silently
    ev = {"type": kind, "ts": int(time.time()), **data}
    async def send():
        try:
            await self.producer.send_and_wait(TOPIC, ev)
        except Exception as e:
            print(f"[events] send failed: {e}") # an error never stops the game
    asyncio.get_running_loop().create_task(send()) # "fire and forget"
```

With `create_task` the send is handed to a separate task; the game loop continues without waiting for the answer. This is called **fire and forget**.

9.3 The consumer: no event is ever lost

```
async for msg in consumer:
    ev = msg.value
    async with db.transaction():
        # both writes or none
        await cur.execute("INSERT INTO game_events(type, ts, data) VALUES (%s, %s, %s)", ...)
        await cur.execute("INSERT INTO event_counts(type, n) VALUES (%s, 1) "
                           "ON CONFLICT (type) DO UPDATE SET n = event_counts.n + 1", ...)
    await consumer.commit()
    # offset committed AFTER the database write
```

At-least-once delivery

If the offset were committed automatically and the worker crashed just before writing to the database, Kafka would consider the message "read" and the event would be lost. Here the commit happens **after** the write. If the worker crashes, it reads the same message again when it restarts. The price: an event can, rarely, be written twice. Writing twice was preferred to losing it.

9.4 What if Kafka is not ready?

Containers are not always ready in the desired order. Both the backend and the worker retry the Kafka connection 30–60 times, every 2 seconds, instead of giving up immediately. The worker also creates the topic itself up front so the consumer group can join straight away.

Watch out: a harmless warning

While Kafka creates its internal topics on first start, the worker logs may show

`GroupCoordinatorNotAvailableError`. It is temporary: a few seconds later the line

`[worker] listening: kafka:9092 / mezarpoly.events` appears and the problem resolves itself.

10 · Introduction to cloud computing and Microsoft Azure

10.1 What is the cloud?

Cloud computing means renting computing power, storage and networking from a provider (Microsoft Azure, Amazon AWS, Google Cloud) over the internet, paying for what you use, instead of buying your own hardware.

Analogy: electricity

You do not build a power plant at home; you plug in and pay for what you use. The cloud does the same for computing power.

10.2 Service models

Model	You manage	The provider manages	Example
IaaS (infrastructure)	Operating system, updates, application	Hardware, network	Azure Virtual Machines
CaaS (containers)	The container image and settings	Servers, OS, scaling	Azure Container Apps
PaaS (platform)	Only the code	The whole runtime	Azure App Service
SaaS (software)	Nothing, you just use it	Everything	Outlook, Teams

10.3 Basic Azure concepts

Concept	Explanation	In this project
Subscription	The billing account; resources belong to a subscription.	Azure for Students (student credit)
Resource group	A folder for related resources; deleting it deletes everything inside.	<code>mezarpoly-rg</code>
Region	The data centre where the resources physically are.	France Central
Resource provider	Enables a service type in the subscription.	<code>Microsoft.App</code> (chapter 13)
Azure CLI (<code>az</code>)	Tool to manage Azure from the command line; everything can be repeated by script.	<code>deploy.sh</code> consists only of <code>az</code> commands.

10.4 Why Azure Container Apps?

Option	Strength	Why not / why yes for this project
Virtual Machine + Docker	Full control; Compose works as is	Runs on one machine — does not meet "each container a separate component"; we maintain the OS.
Azure Kubernetes Service	Industry standard, very powerful	Heavy to learn and run; too much for a small project.
App Service	Very easy for web apps	Not suited to running TCP services like PostgreSQL and Kafka as containers.
Container Apps	The power of Kubernetes, the ease of serverless	Chosen: each container a separate app, internal/external networking, TCP support, pay per use.

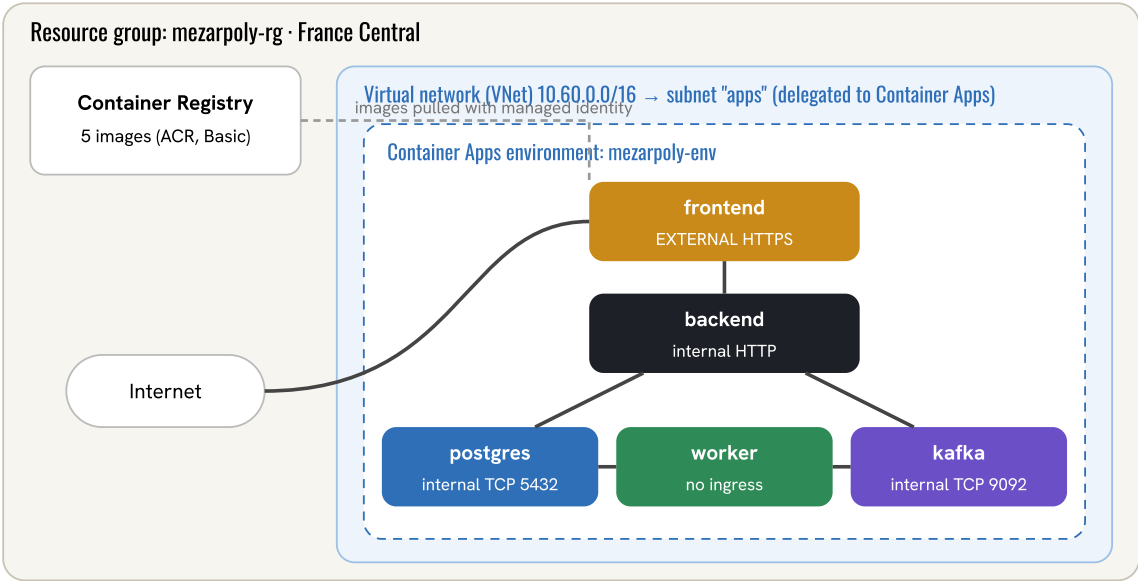


Figure 11.1 — The structure on Azure. Only the frontend can be reached from the internet; the other four apps stay inside the network.

11.1 Azure Container Registry (ACR)

Azure's private image store. The `mezarpoly-frontend`, `mezarpoly-backend` and `mezarpoly-worker` images built on the computer are pushed here. The `postgres` and `kafka` images are imported from Docker Hub into ACR, so Docker Hub's download limits do not apply.

11.2 Virtual network (VNet) and subnet

In Azure you can define your own private network. Container Apps supports **TCP ingress** — PostgreSQL and Kafka speak plain TCP, not HTTP — only in environments that run in your own virtual network. So a virtual network with the address range `10.60.0.0/16` and a subnet **delegated** to Container Apps were created.

11.3 The Container Apps environment

The secure boundary where the apps live together: apps in the same environment can find each other by name. Underneath runs a Kubernetes cluster managed by Microsoft, which you never see. The workload profile chosen was *Consumption* (pay for what you use).

11.4 Container Apps and ingress types

Each container was deployed as a separate **Container App**. **Ingress** defines how an app can be reached:

App	Ingress	How the others reach it
frontend	External HTTPS ; Azure provides the certificate	<code>https://frontend.<env>.francecentral.azurecontainerapps.io</code>
backend	Internal HTTP	<code>http://backend.internal.<environment domain></code> (chapter 13)
postgres	Internal TCP 5432	<code>postgres:5432</code>
kafka	Internal TCP 9092	<code>kafka:9092</code>
worker	None	Nobody reaches it; it only makes outgoing connections.

11.5 Revisions and replicas

A **revision** is an immutable snapshot of an app's settings and image at a moment in time. Changing the image or the settings creates a new revision; you can go back if needed. A **replica** is a running instance of a revision. Here every app runs exactly 1 replica (`--min-replicas 1 --max-replicas 1`), because game rooms are kept in the server's memory and several backend replicas could not share the same room.

11.6 Secrets and managed identity

- **Secrets:** the database password and the session key are not written into environment variables in plain text. They are stored as Container App **secrets** and referenced as `secretref:pgpass`. They are not shown in the portal and do not appear in logs.
- **Managed identity:** no username or password was stored for pulling images from ACR. Each app was given an identity by Azure (`--registry-identity system`) with permission to read from ACR. There is no password to steal.
- **Passwords are random:** `deploy.sh` generates them with `openssl rand -hex 16`. No password is written in the code or in this guide.

12 · deploy.sh: the Azure deployment step by step

The whole Azure setup is collected in one script. The script is **idempotent**: it can be run again and again; it does not recreate existing resources but skips or updates them. Unlike clicking in the portal, the script documents the setup, and the same environment can always be reproduced.

Preparation

```
az login # sign in to Azure in the browser
az provider register -n Microsoft.App # enable Container Apps in the subscription (once)
az provider show -n Microsoft.App --query registrationState # wait until "Registered"
```

Preparing the images

```
az acr login -n <registry name>
docker build -f docker/backend/Dockerfile -t <registry>.azurecr.io/mezarpoly-backend:v2 .
docker push <registry>.azurecr.io/mezarpoly-backend:v2
# the same for frontend and worker; postgres and kafka are imported from Docker Hub:
az acr import -n <registry> --source docker.io/library/postgres:17-alpine --image postgres:17-alpine
```

The seven steps of the script

- 1 Resource group and registry.** `az group create` creates the `mezarpoly-rg` group and `az acr create --sku Basic` the image registry (if they do not exist).
- 2 Virtual network.** The `10.60.0.0/16` network and, inside it, the `apps` subnet (`10.60.0.0/23`) delegated to `Microsoft.App/environments`. A Container Apps environment needs a subnet of at least /23.
- 3 Container Apps environment.** `az containerapp env create ... --infrastructure-subnet-resource-id ... --enable-workload-profiles` places the environment in that subnet. This step takes a few minutes.
- 4 postgres.** Internal TCP ingress (5432), 0.5 vCPU and 1 GiB memory. The password is given as a `secret`. Then the connection string (`postgresql://...@postgres:5432/mezarpoly`) is prepared for the backend and the worker.
- 5 kafka.** Internal TCP ingress (9092), single-node KRaft settings. `KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092` matters: Kafka tells clients "reach me at `kafka:9092`", a name that resolves inside the environment.
- 6 worker and backend.** The worker has no ingress. The backend gets internal HTTP ingress (8095); the database address and the session key are passed as secrets.
- 7 frontend.** The environment's internal domain is queried, `BACKEND_URL=http://backend.internal.<domain>` is computed and the frontend is created with **external** ingress. Then the site's public address is given to the backend as a trusted origin (`MP_ORIGINS`; 13.5).

```
# One of the most important lines of the script (step 7):
DOMAIN=$(az containerapp env show -g "$RG" -n "$ENV_NAME" --query properties.defaultDomain -o tsv)
BACKEND_URL="http://backend.internal.$DOMAIN"
```

Running it and the result

```
TAG=v2 bash docker/azure/deploy.sh
...
Deployed: https://frontend.<env>.francecentral.azurecontainerapps.io
App      State    Ingress
-----
postgres Running False
kafka    Running False
worker   Running
backend  Running False
frontend Running True
```

Only the frontend's ingress is external (`True`); the others are internal (`False`) or have none. That is exactly what the architecture requires.

13 · Problems we hit and how we solved them

This chapter describes problems that really happened during the deployment. Being able to explain how you diagnosed a problem is often more impressive in a presentation than having built the system.

13.1 The "Microsoft.App" resource provider is not registered

Symptom: Container Apps commands failed, saying the subscription was not registered for this service.

Cause: every service type in Azure depends on a "resource provider" that must be enabled in the subscription. In new student subscriptions Container Apps is not enabled by default.

Fix: `az provider register -n Microsoft.App` was run and we waited until the state was `Registered`. This can take minutes; we waited and verified instead of trying to force past it.

13.2 Region restriction in the student subscription

Symptom: creating resources was refused in some regions.

Cause: Azure for Students comes with a policy that allows resources only in certain regions.

Fix: **France Central**, which is allowed and supports Container Apps, was chosen (`LOC=francecentral`).

13.3 TCP ingress for PostgreSQL and Kafka

Symptom: TCP ingress could not be defined in the default Container Apps environment.

Cause: Container Apps supports non-HTTP (TCP) ingress only in environments deployed into your own virtual network (*custom VNet*).

Fix: a virtual network and a /23 subnet delegated to Container Apps were created first, and the environment was placed inside it (`deploy.sh` steps 2 and 3).

13.4 The site opens but the game does not: "404 page not found"

The most instructive bug

The home page loaded, but `/healthz`, `/api/...` and the game's WebSocket received `404 page not found`.

Diagnosis:

- The response text was not nginx's error page but that of Azure's internal router (Envoy). So nginx was forwarding the request, but the request never reached the backend.
- The frontend had `BACKEND_URL=http://backend`. Locally Docker's name resolver turns this short name into the backend container. On Azure, internal HTTP traffic passes through a layer that routes by the Host header, and that layer only recognises the app's **full internal name**.

Fix: the environment's domain was queried and the frontend was given the full internal name:

```
BACKEND_URL=http://backend.internal.<environment domain>
```

The fix was not only applied by hand; it was written into `deploy.sh`, the nginx configuration file and the documentation, so re-running the script does not bring the problem back.

The lesson

PostgreSQL and Kafka could be reached by short name (`postgres:5432`) while the backend could not, because TCP traffic is forwarded directly whereas HTTP traffic is routed by the Host header. Even in the same environment, the two kinds of traffic behave differently.

13.5 Logins and game requests rejected (CSRF protection)

Cause: for security, the game server only accepts POST and WebSocket requests from addresses it knows (*Origin* check). This stops another site from sending requests on the user's behalf (CSRF). The Azure address was not known in advance, so at first it was not on the trusted list.

Fix: after the frontend was created, its public address was read and given to the backend as `MP_ORIGINS`, `MP_PUBLIC_URL` and `MP_API_URL` (the last step of the script).

13.6 The user's real IP

On Azure, traffic passes Azure's own load balancer before nginx, so the IP connecting to nginx is the balancer's, not the user's. Since the game's rate limits and anti-abuse rules depend on the IP, nginx was configured to pass the first IP in the `X-Forwarded-For` header to the backend as the real user IP.

14 · Verification: proof that it works

After the deployment the system was tested end to end: the public address was opened, a guest game against bots was started, and the logs confirmed that the event reached the worker through Kafka.

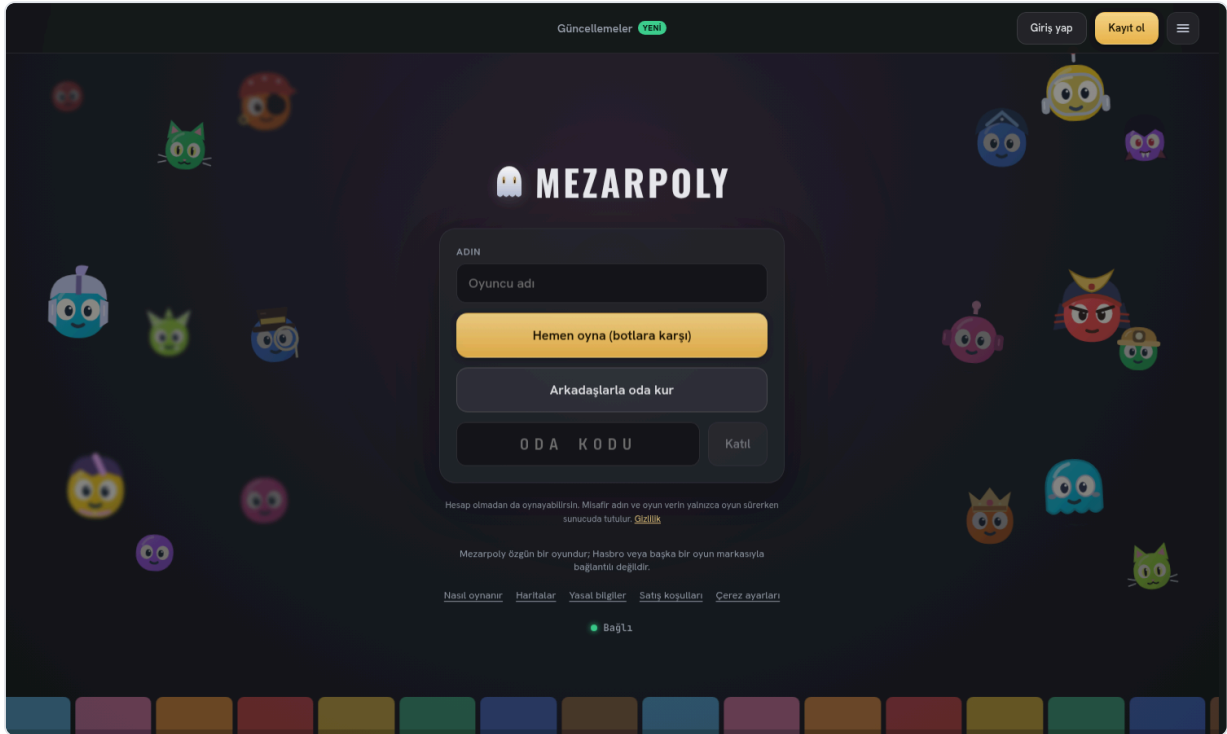


Figure 14.1 — The home page served from the public Azure address (connection indicator: connected).

The state of the apps

App	State	External	Transport	Image
postgres	Running	False	Tcp	<registry>.azurecr.io/postgres:17-alpine
kafka	Running	False	Tcp	<registry>.azurecr.io/kafka:3.9.1
worker	Running			<registry>.azurecr.io/mezarpoly-worker:v2
backend	Running	False	Http	<registry>.azurecr.io/mezarpoly-backend:v2
frontend	Running	True	Auto	<registry>.azurecr.io/mezarpoly-frontend:v2

The event's journey in the logs

```
# backend: connected to Kafka
[events] connected to Kafka: kafka:9092 → mezarpoly.events
# worker: topic ready, listening
[worker] topic created: mezarpoly.events
[worker] listening: kafka:9092 / mezarpoly.events
# the game started → the event went through Kafka into the database
[worker] processed: game_started 2RYA9
```

The last line shows that the "game started" event of room `2RYA9` travelled backend → Kafka → worker and was stored in PostgreSQL. It proves that grade 5.0 (asynchronous communication) and grade 6.0 (every container a separate cloud component) work together.

Reading the logs

```
az containerapp logs show -g mezarpoly-rg -n worker --tail 50
az containerapp logs show -g mezarpoly-rg -n backend --tail 50
```

15 · Cost, stopping and restarting

15.1 What do we pay for?

Resource	How it is billed	When stopped
Container Apps (5 apps)	vCPU and memory time of running replicas	0 replicas → no compute cost
Container Registry (Basic)	A small fixed daily fee + storage	Continues until deleted
Virtual network	Free for basic use	—

15.2 Stop and start scripts

```
bash docker/azure/stop.sh    # deactivates every app's active revision → 0 replicas
bash docker/azure/start.sh    # re-activates the revisions (1-2 minutes)
```

Watch out: data is not persistent

In this setup postgres and kafka have no persistent disk. When the apps are stopped and started, the database and the events are reset. That is enough for the assignment demo; a real system should use Azure Database for PostgreSQL (managed database) and Azure Event Hubs (a managed service compatible with Kafka) or Azure Files volumes.

15.3 Deleting everything

```
az group delete -n mezarpoly-rg --yes --no-wait    # deletes EVERYTHING in the resource group
```

15.4 Ideas for improvement

- Persistent data: move to managed PostgreSQL and Event Hubs.
- Continuous deployment: GitHub Actions builds the images and deploys to Azure on every `git push`.
- Monitoring: central logs and metrics with Log Analytics and Application Insights.
- Scaling: move room state to a shared store like Redis and run several backend replicas.

16 · Live demo script for the presentation

Suggested length: 12 minutes in three parts (Azim, Gloria, Kutay — about 4 minutes each), then questions. Run `start.sh` 10 minutes before the presentation.

- 1 Show the architecture (Azim).** Figure 6.1: five containers, two networks, solid and dashed arrows.
- 2 Show it locally (Gloria, optional).** `docker compose ps` for the five services and their health; `docker network inspect` to prove the frontend is not on the database network.
- 3 Show the Azure portal (Kutay).** The five Container Apps in the `mezarpoly-rg` resource group and their ingress types (external / internal / TCP / none).
- 4 Play the game (Kutay).** Open the public address, press "Play now", roll the dice. Keep `/status.html` open in another tab; the new event appears within seconds.
- 5 Prove Kafka (Kutay).** Show the `processed: game_started` line in the output of `az containerapp logs show ... -n worker`.
- 6 Tell one problem (Kutay).** The 404 error from 13.4, its diagnosis and fix. This usually draws the most questions.

Backup plan

If the internet or Azure has a problem, the same flow can be shown with the local Compose version and the screenshots in this guide.

17 - Command cheat sheet

Docker / Compose

Command	What it does
<code>docker compose up --build -d</code>	Builds the images, starts everything in the background
<code>docker compose ps</code>	Shows the services' state and health
<code>docker compose logs -f backend</code>	Follows one service's logs live
<code>docker compose exec postgres psql -U mezarpoly</code>	Opens an SQL client in the database container
<code>docker compose down</code> / <code>down -v</code>	Stops / stops and deletes the data
<code>docker images</code> , <code>docker ps</code>	Lists images / running containers
<code>docker network ls</code> , <code>docker network inspect <name></code>	Lists networks / shows the containers on one

Kafka (from inside the container)

```
docker compose exec kafka /opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --
topic mezarpoly.events
docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --
describe --group mezarpoly-stats
# LAG = 0 → the worker has processed every event
```

PostgreSQL

```
docker compose exec postgres psql -U mezarpoly -d mezarpoly -c "SELECT id, type, data FROM game_events
ORDER BY id;"
curl http://localhost:8080/api/events/summary
```

Azure CLI

Command	What it does
<code>az login</code>	Signs in to Azure
<code>az provider register -n Microsoft.App</code>	Enables Container Apps in the subscription
<code>az containerapp list -g mezarpoly-rg -o table</code>	Lists the apps
<code>az containerapp logs show -g ... -n worker</code>	Shows one app's logs
<code>az containerapp revision list -g ... -n backend</code>	Lists revisions
<code>az containerapp update -g ... -n frontend --set-env-vars K=V</code>	Changes an environment variable (new revision)
<code>az acr repository list -n <registry></code>	Lists the images in the registry
<code>az group delete -n mezarpoly-rg</code>	Deletes everything

18 · Glossary

ACR (Azure Container Registry)

Azure's private image store.

aiohttp

Asynchronous web server and client library for Python; the basis of the backend.

At-least-once

Delivery where no message is lost but one may rarely be processed twice.

Broker

The Kafka server that stores and distributes messages.

CLI

Command-line interface (e.g. `az`, `docker`).

Compose

Docker tool that manages several containers from one file.

Container

A running, isolated copy of an image.

Container App

A single containerised app in Azure Container Apps.

CSRF

An attack where another site sends unwanted requests with the user's session.

DNS

The system that turns names (`postgres`, `mezarpoly.fun`) into IP addresses.

Dockerfile

The recipe that builds an image.

Envoy

The proxy that routes internal traffic in Container Apps.

FQDN

Fully qualified domain name, e.g. `backend.internal.<env>`.

Health check

A command that regularly tests whether a container is really ready.

Idempotent

An operation that gives the same result however often it runs.

Image

An immutable package of an application and its dependencies.

Ingress

The rule for reaching an app from outside (external, internal, TCP).

JSON

A common text format for data.

Kafka

A distributed event-streaming and messaging platform.

KRaft

Kafka's mode that does not need ZooKeeper.

Managed identity

A password-less identity Azure gives to a resource.

nginx

Web server and reverse proxy.

Offset

The sequence number of a message in a Kafka partition.

Port

The numbered door a program listens on.

PostgreSQL

An open-source relational database server.

Replica

A running instance of a revision.

Resource group

A folder for related resources in Azure.

Reverse proxy

A front door that distributes requests to servers behind it.

Revision

An immutable version snapshot of a Container App.

Secret

A hidden value used by reference instead of plain text.

SQLite

A single-file embedded database.

Subnet

A slice of a virtual network's address range.

Tag

An image's version name (`v2`).

Volume

Persistent storage independent of a container.

VNet

A private virtual network defined in Azure.

WebSocket

A permanent two-way connection between browser and server.

Worker

A process that does queued work in the background.

19 · Test yourself: 25 questions and answers

Try to say the answer yourself first, then read it.

1. What is the main difference between a container and a virtual machine?

A virtual machine carries its own operating system. A container shares the host's kernel and contains only the application and its dependencies, so it is smaller and faster.

2. How are an image and a container related?

An image is an immutable template; a container is a running copy of it. Many containers can run from one image.

3. Why is requirements.txt copied before the source code in a Dockerfile?

For the layer cache: when code changes, the library layer does not, so libraries are not reinstalled.

4. What is the benefit of a multi-stage build?

Build tools (Node.js) do not end up in the final image, which stays small with a smaller attack surface.

5. Why do we run containers as a non-root user?

So that if a flaw is exploited, the attacker's rights stay limited ("least privilege").

6. Why does this project have two Docker networks?

So that the internet-facing frontend has no direct access to the database and Kafka. Only the backend reaches them.

7. How does the backend reach postgres — how does it know the IP address?

It does not need to. Containers on the same network find each other by service name (`postgres`), which Docker's DNS resolves.

8. Why is depends_on alone not enough — why service_healthy?

`depends_on` only sets the start order; it does not wait until a service is ready. `service_healthy` waits until the health check passes.

9. Which headers does nginx need for WebSocket?

`Upgrade` and `Connection`, plus HTTP/1.1. For long games `proxy_read_timeout` was also raised.

10. Give one example each of synchronous and asynchronous communication from this project.

Synchronous: logging in, loading the leaderboard. Asynchronous: writing game started/finished events to Kafka.

11. What happens to the game if Kafka goes down?

Nothing; publishing is fire-and-forget, the error is only logged and the game continues.

12. Why does the worker commit the offset after the database write?

So that the message is not lost if it crashes in between (at-least-once delivery).

13. What is the cost of at-least-once delivery?

An event can, rarely, be processed twice.

14. What is KRaft?

Kafka's mode without ZooKeeper: one container acts as both broker and controller.

15. How does the same code work with SQLite and PostgreSQL?

The data layer is written once against the `sqlite3` interface; a thin adapter (`pgcompat.py`) offers that interface for PostgreSQL and translates the dialect. Our test suite runs against PostgreSQL to prove it.

16. Why Azure Container Apps?

It makes every container a separate component, offers internal/external networking and TCP support, needs no Kubernetes management, and you pay for what you use.

17. Why was a custom virtual network (VNet) needed?

Container Apps supports the TCP ingress that PostgreSQL and Kafka need only in environments inside a custom VNet.

18. Which app is open to the internet, and why only that one?

Only the frontend. The others have internal ingress or none, which keeps the attack surface small.

19. Why could the backend not be reached by its short name on Azure?

Internal HTTP traffic is routed by the Host header and only recognises the full internal name (`backend.internal.<domain>`). TCP apps can be reached by short name.

20. How was the database password stored?

It was generated randomly and stored as a Container App secret, passed to the environment variable with `secretref: .`

21. How do the apps pull images from ACR without a password?

With a system-assigned managed identity that was given read permission on ACR.

22. What is a revision and why is it useful?

An immutable snapshot of an app version. Every change creates a new revision; you can roll back if needed.

23. Why does the backend run as a single replica?

Game rooms are kept in the server's memory; several replicas could not share the same room. To scale, the state must move to a shared store.

24. What was done to avoid spending credit?

All revisions were deactivated with `stop.sh` (0 replicas). The registry and network remain; `start.sh` brings everything back in 1-2 minutes.

25. How was it proven that the system works?

A game was played through the public address; the `processed: game_started` line in the worker log shows that the event completed the backend → Kafka → worker → PostgreSQL path.

Source code and details: `docker/README.md`, `docker-compose.yml`, `docker/azure/deploy.sh` in the project repository. Game: `mezarpoly.fun` · Team: Azim, Gloria, Kutay · `kutayonur.dev/azure`