

Part 1:

The application and Docker

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

Azim

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 open the presentation. Your job is to make the audience understand **what the system is** before anyone talks about the cloud: the game, the requirements, the five containers, and how the container images are built.

Slide	Topic	Time
1	Title and team introduction	0:20
2	Part 1 divider	0:05
3	Agenda: three parts, three presenters	0:20
4	The application: Mezarpoly	0:35
5	Grade map: requirements and how we meet them	0:40
6	Architecture: five containers, two networks	0:50
7	What runs where (container table)	0:30
8	Three good habits in the Dockerfiles	0:40
	Total	≈ 4:00

Your three key messages

- Mezarpoly is **our own live application**, not a toy example, and we designed it from day one as five containers.
- The system is **five containers on two networks**; the internet-facing frontend cannot reach the database.
- The images follow **good practice**: multi-stage build, non-root user, layer caching, health checks.

How you hand over

At the end of slide 8 say: *"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."* Then stop sharing your screen so Gloria can share hers. **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.

Title — Five containers, one cloud



WHAT TO SAY

Good morning. We are Azim, Gloria and Kutay.

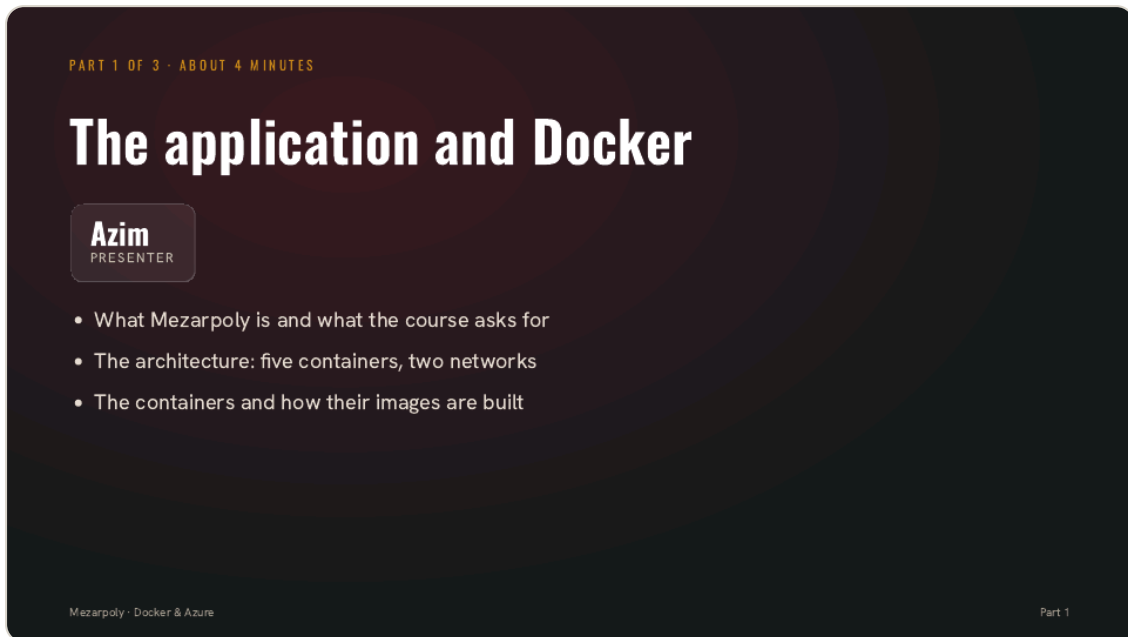
Our project takes **Mezarpoly**, a browser-based multiplayer property-trading game that real people play at mezarpoly.fun, and turns it into a containerised system that runs in the cloud.

In the next twelve minutes we will show the five containers, how they talk to each other, how every one of them runs as a separate component on Microsoft Azure, and what went wrong on the way.

REMEMBER

- Speak slowly and look at the audience, not at the screen.
- Introduce the team by name: this tells the examiner that three people will speak.

Part 1 divider — The application and Docker



WHAT TO SAY

I will start with the application itself and how we packaged it with Docker.

REMEMBER

- Do not stay on this slide; one sentence and move on.

Agenda — three parts, three presenters

The slide is titled "Three parts, three presenters" and is labeled "PART 1 - AZIM" in the top right corner. It features three columns, each representing a part of the agenda and its presenter. The first column, titled "1 - Azim", lists topics: "The application and the requirements", "the architecture", "the containers and their images". The second column, titled "2 - Gloria", lists topics: "Docker Compose", "the synchronous path", "Kafka and the worker", "PostgreSQL", "why Azure Container Apps", "the Azure architecture". The third column, titled "3 - Kutay", lists topics: "The deploy script", "security decisions", "problems and fixes", "proof that it works", "cost and next steps", "questions". The footer of the slide reads "Mezarpoly · Docker & Azure" and "3".

1 - Azim	2 - Gloria	3 - Kutay
The application and the requirements · the architecture · the containers and their images	Docker Compose · the synchronous path · Kafka and the worker · PostgreSQL · why Azure Container Apps · the Azure architecture	The deploy script · security decisions · problems and fixes · proof that it works · cost and next steps · questions

WHAT TO SAY

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 really runs.

REMEMBER

- Point at each column as you name it.

The application — Mezarpoly

THE APPLICATION

Mezarpoly

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



Mezarpoly · Docker & Azure

PART 1 · AZIM

- 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

4

WHAT TO SAY

Mezarpoly is the real-time board game the three of us built, for two to eight players. You roll dice, buy cities, build houses and hotels, trade with other players and bid in auctions.

The game itself runs over a **WebSocket**: a connection that stays open, so when one player rolls the dice, every other player's screen updates immediately.

Around the game there are accounts, a shop, friends, leaderboards and a ranked league.

We designed it from the beginning as **five services**: a web front, the game server, a database, a message broker and a worker, each switched on by configuration. When we launched the public site 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 planned.

The screenshot on the slide was taken from our Azure deployment.

REMEMBER

- "Not a toy example" is a strong point: say it clearly.
- If asked what a WebSocket is: a long-lived, two-way connection; the server can push messages without the browser asking.

Grade map

REQUIREMENTS
PART 1 · AZIM

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.

Mezarpoly · Docker & Azure
5

WHAT TO SAY

This table maps every grade step of the course to the part of the project that meets it.

For **3.0**, one command — `docker compose up` — starts the application.

For **4.0**, two containers communicate over a Docker network: the nginx frontend forwards HTTP and WebSocket traffic to the backend.

For **4.5**, a third container is the database: PostgreSQL stores accounts, game results, the shop and friends.

For **5.0**, a fourth container handles asynchronous messaging: game events go through Kafka to a worker that writes them to the database.

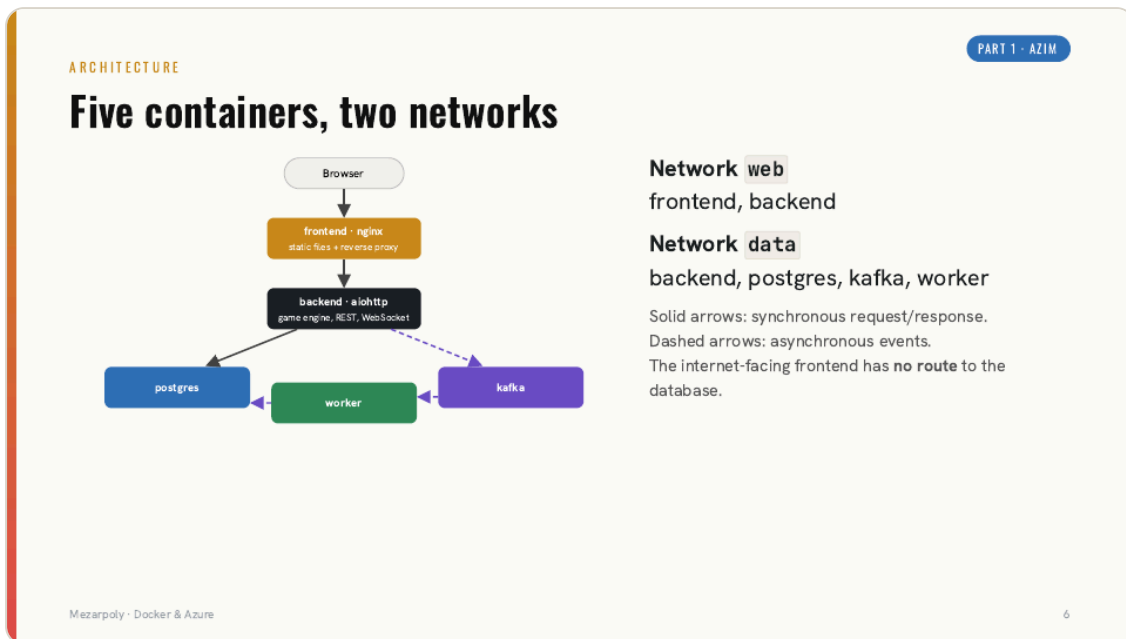
And for **6.0**, all containers run in the cloud as separate components: each one is its own Azure Container App.

Each step builds on the previous one, so the rest of our talk follows this table from top to bottom.

REMEMBER

- Read the grade numbers out loud; examiners like to hear the mapping.
- Do not explain Kafka or Azure in detail here — Gloria and Kutay will.

Architecture — five containers, two networks



WHAT TO SAY

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, login and WebSocket traffic to the **backend**, our Python game server.

The backend reads and writes **PostgreSQL** directly — that is the synchronous path, the solid arrows.

When a match starts or ends, the backend also publishes an event to **Kafka** — the dashed purple arrows. A separate **worker** reads those events and stores them in PostgreSQL. That is the asynchronous path.

There are **two Docker networks**. The frontend is only on the *web* network; the database, Kafka and the worker are only on the *data* network. The backend is the only bridge. So even if someone broke into nginx, they would have no route to the database. This is called network segmentation.

REMEMBER

- Trace the arrows with your hand while speaking.
- Key phrase: "the frontend has no route to the database".
- Solid arrows = synchronous (wait for an answer); dashed = asynchronous (leave a message and continue).

What runs where

CONTAINERS

PART 1 · AZIM

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 mezarpoLy.events
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.

Mezarpoly · Docker & Azure

7

WHAT TO SAY

Three of these images are built from our repository — frontend, backend and worker — and two are official images: PostgreSQL 17 and Kafka 3.9.

Kafka runs in **KRaft** mode, so it does not need a separate ZooKeeper container: one container is both the broker and the controller.

All four long-running services have **health checks**. That matters for the start-up order, which Gloria will show.

REMEMBER

- Ports to remember: nginx 80, backend 8095, PostgreSQL 5432, Kafka 9092.

Three good habits in the Dockerfiles

DOCKER IMAGES

PART 1 - AZIM

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)"
```

Mezarpoly · Docker & Azure

8

WHAT TO SAY

We applied three good habits when writing the Dockerfiles.

First, a **multi-stage build** for the frontend. In the first stage a Node.js image compiles the game's templates; only the finished files are copied into the final nginx image. The build tools never reach production, so the image is smaller and has less to attack.

Second, the Python containers run as a **non-root user**. If someone found a bug in the program, they would have very few rights inside the container.

Third, **layer caching**: we copy and install the libraries before the source code. When we change one line of code, Docker reuses the library layer and the build takes seconds.

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."

REMEMBER

- Multi-stage = smaller and safer image.
- Non-root = least privilege.
- Layer cache = order of instructions matters.

Background you should understand

Container vs. virtual machine

A **virtual machine** imitates a whole computer with its own operating system; it is heavy (gigabytes) and slow to start. A **container** shares the host's operating-system kernel and only packages the application and its libraries; it starts in seconds and is small. That is why containers are the standard way to ship web applications.

Image, container, registry

Term	Meaning
Image	A frozen, read-only package of the application and everything it needs (like a recipe or a mould).
Container	A running instance of an image. Many containers can run from one image.
Dockerfile	The text file with the instructions that build an image.
Registry	A store for images: Docker Hub, Azure Container Registry.
Tag	The version label of an image, e.g. <code>mezarpoly-backend:v2</code> .
Volume	Persistent storage that survives when a container is removed.

Layers and the build cache

Every instruction in a Dockerfile creates a **layer**. When you rebuild, Docker reuses every layer whose inputs did not change. If you copied the source code before installing libraries, every code change would invalidate the library layer and reinstall everything. That is why the order is: base image → copy requirements → install → copy code.

The backend Dockerfile, line by line

```
FROM python:3.13-slim          # small Debian with Python 3.13
WORKDIR /app
COPY requirements.txt docker/backend/requirements-docker.txt ./
RUN pip install --no-cache-dir -r requirements.txt -r requirements-docker.txt \
    && useradd -r -u 10001 app && mkdir -p /data && chown app /data
COPY server/ ./server/        # code last → cache
USER app                      # not root
ENV MP_HOST=0.0.0.0 MP_PORT=8095 ... # listen on all interfaces
EXPOSE 8095
HEALTHCHECK ... /healthz
CMD ["python", "server.py"]
```

`0.0.0.0` means "listen on every network interface"; without it the server would only accept connections from inside its own container.

The frontend multi-stage build

```
# Stage 1: Node 22 compiles the Vue templates and assembles the static site
FROM node:22-alpine AS build
...
RUN bash tools/dist.sh
# Stage 2: only nginx + the finished files
FROM nginx:1.29-alpine
COPY --from=build /src/dist/ /usr/share/nginx/html/
```

`COPY --from=build` is the key line: it takes files from the first stage. Everything else in that stage (Node.js, npm, compilers) is thrown away.

Two networks — why it matters

Containers on the same Docker network find each other by **service name** (Docker runs an internal DNS). Containers on different networks cannot see each other at all. The frontend is the only container exposed to the internet, so it is the most likely to be attacked; keeping it off the data network means an attacker who controls nginx still cannot open a connection to PostgreSQL. This is the network form of the **principle of least privilege**.

Synchronous vs. asynchronous

Synchronous: you ask and wait for the answer (a phone call) — e.g. logging in. **Asynchronous:** you leave a message and carry on; it is processed later (a note on a notice board) — e.g. "game finished" events going to Kafka. The game never waits for statistics to be written.

One code base, two modes

We wrote the game so that the container parts are activated by environment variables: `MP_DATABASE_URL` selects PostgreSQL, `MP_KAFKA` switches on event publishing. With them it is the full five-container system; without them it is the light single-server mode the public site uses. This is the "configuration separate from code" principle.

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. Why did you split the application into five containers instead of one?

Each container has one job, which makes the system easier to understand, replace and scale. It also lets us isolate the database from the internet-facing part, and the course requires separate components for the database and the messaging system.

2. What is the difference between an image and a container?

An image is a read-only template; a container is a running instance of it. We can start several containers from one image.

3. Why is the frontend nginx and not the Python server directly?

nginx is very efficient at serving static files and acts as a single entry point (reverse proxy). It forwards only /api, /auth and /ws to the backend, so the backend never has to serve static files and is not directly exposed.

4. What does a multi-stage build give you?

A smaller and safer final image: the build tools (Node.js, npm) stay in the first stage and never reach the image that runs in production.

5. Why run as a non-root user?

Least privilege: if the process is compromised, the attacker cannot change system files or install software inside the container.

6. What is a health check, and what happens if it fails?

A command Docker runs periodically (for the backend: an HTTP request to /healthz). After several failures the container is marked unhealthy; Compose will not start services that depend on it being healthy.

7. How do the containers find each other?

By service name, through Docker's internal DNS on each network — for example the backend connects to postgres:5432. No IP addresses are hard-coded.

8. Why two networks?

So that the internet-facing frontend has no route to the database and Kafka. Only the backend is on both networks.

9. What is KRaft?

Kafka's mode without ZooKeeper: the Kafka process also acts as the controller that manages the cluster metadata, so one container is enough.

10. Why does the public site not run all five containers?

At its current size it does not need them. We designed the game for five containers from the start and every part is switched on by configuration, so the public site runs the same code in a light single-server mode, and the course deployment runs the full design.

11. What is a WebSocket and why does the game need it?

A long-lived two-way connection. The server can push updates to every player instantly, which a turn-based multiplayer game needs.

Terms in your part

Container

An isolated running instance of an image.

Image

Read-only package of an application and its dependencies.

Dockerfile

Instructions that build an image.

Layer

One step of an image; cached between builds.

Multi-stage build

Building in one stage and copying only the result into a clean final stage.

Registry

Where images are stored (Docker Hub, ACR).

Health check

A periodic test of whether a container is ready.

Reverse proxy

A front door that forwards requests to back-end servers (nginx).

WebSocket

Persistent two-way browser-server connection.

Network segmentation

Splitting a system into networks so parts cannot reach each other unnecessarily.

Least privilege

Give each component only the rights it needs.

KRaft

Kafka without ZooKeeper.

Port

Numbered door a program listens on (80, 8095, 5432, 9092).

SQLite

Single-file embedded database used in the light single-server mode.

Checklist before you present

- ☐ Rehearse your part out loud twice with a timer (target: 4 minutes).
- ☐ Know the five container names and their ports by heart.
- ☐ Be able to explain the two networks without looking at the slide.
- ☐ Practise the handover sentence to Gloria.
- ☐ Open Slides-Part1-Azim.pdf full screen and test screen sharing in the call app before the presentation.
- ☐ Turn off notifications (do not disturb) and close other windows before you share.
- ☐ Have the architecture slide (6) ready to jump back to if questions come later.