1. Intro / Goal

“Let me walk you through our deployment automation.
We use Ansible to configure a remote Ubuntu VM, install everything it needs, deploy our code, start our services, and verify they’re working.

---

2. Inventory (which machine are we deploying to?)

“In Ansible, the ‘inventory’ file is just a list of servers we control.

Ours says:

- we have a host called `ronin-vm`
- its address is `minibus-test-phong.uoa.cloud`
- connect as the `ubuntu` user
- and use my SSH private key to authenticate.

So basically: this tells Ansible *where to log in* and *as who*.

Then, in the playbook, I say `hosts: ronin`, which means: run all these steps on that VM.”

(You can mention: “So step one is: define the target. We call that the inventory.”)

---

3. Start of the playbook

“The playbook starts with:

```
- name: Deploy Cyber_Minibus to Ronin
  hosts: ronin
  become: yes
  gather_facts: yes
```

- `name` is just a description.
- `hosts: ronin` means run on the ronin group from the inventory.

- `become: yes` means use sudo when needed, because we're installing system stuff like Docker and systemd services.

- `gather_facts: yes` means Ansible will detect OS info on the remote machine, like 'what Ubuntu version is this', 'what architecture', etc. Later tasks can use that info dynamically."

You can say:
"This matters because we may have slightly different steps depending on OS version. We don't hardcode 'it's Ubuntu 20', we let Ansible detect it."

---

4. Variables

"In the playbook we also define variables. For example:

- `jwt_secret` – this is our backend auth secret (we can override this at runtime with `-e` so we don't hardcode secrets in the repo).

- `vite_base_url` – what URL the frontend should use to call the backend API.

- toggles like `client_build`, `server_vulnerable_mode`, etc. These change behavior without changing code.

- `project_dir` – where on the VM the app will live, for example `/home/ubuntu/Cyber_Minibus`.

- `repo_url` – the Git repo we're deploying from.

- `mongodb_port` – which port MongoDB should listen on (27017).

So these variables make the playbook reusable.
If we deploy to a different box or change where the code lives, we just change the vars, not the tasks."

---

5. Base system setup (apt update, required packages)

"Then we start configuring the VM.

First, we update apt and install some basic packages we'll need later: things like `curl`, `git`, `unzip`, `gnupg`, etc.

Why? Because later tasks depend on them:

- we use `curl` to grab keys and install Docker,
- we use `git` to pull our application code from the repo,
- we use `gnupg` to verify package repositories.

So this is like ‘prep the box so it can install bigger stuff.’”

---

6. Install Node.js

“We need Node.js because:

- the backend is a Node/Express server,
- the frontend (Vite) is a Node-based build tool.

So in the playbook we:

1. Add the NodeSource repository for Node 20,
2. Install `nodejs` with apt.

That guarantees we’re running a known Node version on the VM, not whatever random version came with Ubuntu.”

You can say:
“This gives us a consistent runtime for both server code and frontend build steps.”

---

7. Install Docker and Docker Compose

“This is a big one.

We install Docker on the VM so we can run MongoDB in a container instead of installing MongoDB directly on the host.

The Ansible steps do the following:

- add Docker’s GPG key and apt repo

- `apt install docker-ce docker-ce-cli containerd.io`
- install the `docker-compose-plugin` so we can run `docker compose up`
- enable and start the Docker service
- add our `ubuntu` user to the `docker` group

That last step is important: it means we can run `docker` without typing `sudo` every time.

In plain English: after this part, the VM is Docker-ready."

---

8. Make sure the project directory exists

"Next, we create the directory on the VM where our application will live.
For example: `/home/ubuntu/Cyber_Minibus`.

We set the owner to our non-root user so we don't have to sudo for every file write.

This is prepping a home for our code."

---

9. Clone (or update) the Git repo onto the VM

"We then pull the latest code from our Git repo into that project directory.

If it's already cloned, we update it (so this works for redeploys too).
If it isn't there yet, we clone fresh.

That guarantees the VM is always running the latest commit of our app."

---

10. Generate `.env` files (configuration for server and client)

"We template out the environment files.

There are tasks that say:

- copy `templates/server.env.j2` to `server/.env`

- copy `templates/client.env.j2` to `client/.env`

This is how we inject environment-specific values:

- secrets like the JWT secret,
- API base URLs,
- currency symbol,
- vulnerability mode toggles,
- etc.

This is important because the app needs config to boot, and production config is not always the same as local dev config."

You can say:
"This step is: 'teach the app where it's running and with what secrets.'"

---

11. Install Node dependencies for server and client

"The playbook then runs `npm ci` or `npm install` in two places:

- `server/` (backend)
- `client/` (frontend)

Why two? Because those are two separate Node projects.

We're making sure all required Node modules are available on the VM so the backend can run and so we can build the frontend if we choose to build it."

You can add:
"We run these commands as the normal user, not root, which avoids permission nightmares later."

---

12. (Optional) Build the frontend

"We have a toggle called `client_build`.

If `client_build` is true, we run `npm run build` in the `client/`.
That produces a production build in `client/dist/` — static HTML/CSS/JS.

Those built files are what we would later serve with Nginx in production on port 80.

If `client_build` is false, we skip this, which is useful for dev or for testing."

Explain it as:
"This is where we turn developer code into deployable frontend assets."

---

13. Bring up MongoDB using Docker Compose

"Now we set up the database.

Instead of installing MongoDB directly on the VM, we rely on Docker Compose.
We have a `docker-compose.yml` in the project that defines a `mongo` service.

The playbook runs:

```
docker compose up -d
```

in the project directory.

That:

- pulls the MongoDB image,
- creates a named volume for persistent data,
- runs MongoDB in the background (detached),
- maps port 27017.

This gives our app a database to talk to."

You can add:
"This is one of the main reasons Docker is in this deployment at all."

---

14. Wait for MongoDB to be ready

"Right after starting Mongo, we don't just assume it's ready instantly.

We have a `wait_for` task:

- it waits for port 27017 on localhost to be open,
- for up to X seconds.

This is important because the next step depends on MongoDB being available.
If Mongo isn't up and responsive yet, seeding would fail."

So you can say:
"This is us being patient and reliable instead of racing ahead."

---

15. Seed the database (first-time data load)

"Then we run our seed script:

```
npm run seed
```

This does things like:

- connect to MongoDB,
- clear old data,
- insert dummy or initial data.

After a successful seed, we 'touch' a marker file (for example `.minibus_seeded`) so that if we re-run the playbook, it won't keep reseeding and wiping data every time.

So: first deploy — it seeds.
Future deploys — it skips."

That's important to say in a demo:
"This prevents us from nuking production data accidentally."

---

16. Install a systemd service for the backend

"Now that MongoDB is running and seeded, we make the backend API run as a real service.

We install a systemd unit file (like `/etc/systemd/system/minibus-server.service`) using a template.

That unit file basically says:

- WorkingDirectory = `/home/ubuntu/Cyber_Minibus/server`
- ExecStart = `/usr/bin/node server.js`
- Restart = always
- User = ubuntu

Then we reload systemd, enable it, and start it.

What that means in plain English:
The backend is now managed like any other Linux service:

- it starts automatically on boot,
- if it crashes, systemd restarts it,
- we can check status with `systemctl status minibus-server`."

This is huge. Say this clearly:
"This is where our app stops being 'a Node script I run manually' and becomes 'a service running 24/7 on the server.'"

---

17. Health check / verification

"Finally, we do a quick check:

We `curl http://localhost:3000` on the VM and store the output.

That's a smoke test:

- Is the backend responding on port 3000?
- Did Node boot, read .env, connect to Mongo?

Then we print that result with an Ansible `debug` task.

So at the end of the run, we know if the app is alive."

You can phrase it as:
"This is our 'did we just deploy something real, or is it on fire' moment."