You built a *mini production-like* system made of small services that interact. The goal was to demonstrate observability (metrics → Prometheus), visualization (Grafana), alerting (Alertmanager), and automation (alert bot that restarts/scales services). We used Docker Compose to wire everything together so it's reproducible on a single machine.

---

1. Fundamentals — the core technologies

Docker (images, containers, compose)

- **What it is:** Docker packages software (and its runtime) into *images* and runs them as *containers* — isolated lightweight processes with their own filesystem and networking.
- **Why use it:** Reproducibility. Everyone runs the same image, same environment. Easier to spin up multi-service demos locally.
- **Key concepts:**
  - *Image* — a snapshot (like a VM template).
  - *Container* — a running instance of that image.
  - *Volumes* — persisted data (e.g., Postgres data).
  - *Networks* — Docker provides internal service DNS so api can reach postgres_db by name.
  - *Docker Compose* — a YAML file that declares multiple services and links them; great for local multi-container stacks.

Flask (the API)

- **What it is:** A minimal Python web framework used to build HTTP APIs.
- **Why Flask for the demo:**
  - Extremely simple to set up and readable.
  - Fast to instrument for metrics (prometheus-flask-exporter).
  - Shows typical API behavior without heavy boilerplate.
- **Alternatives:** FastAPI (faster, built-in async), Django (full-featured). For a short demo, Flask is the simplest.

PostgreSQL (the DB)

- **What it is:** A reliable relational database (ACID, SQL).
- **Why Postgres for the demo:**
  - Realistic: many production apps use relational DBs.
  - Supports complex queries and shows realistic DB metrics/latency.

    - Better to test DB-related failures and alerts than an in-memory store.
- **Alternatives:** MySQL, SQLite (not realistic), NoSQL stores—Postgres offers good demo fidelity.

Prometheus (monitoring)

- **What it is:** Time-series metrics database and *pull-based* scraper. Prometheus scrapes /metrics endpoints.
- **Why Prometheus:**
    - Widely used in production for metrics.
    - Pull model (scrape) simplifies collection from many ephemeral services.
    - Strong PromQL language for alerts and queries.
- **Key ideas:** Metrics are *scraped* at intervals; they are typically counters/gauges/histograms.

Exporters (node_exporter, blackbox_exporter)

- **node_exporter:** exports host/container OS metrics (CPU, memory, disk).
- **blackbox_exporter:** probes external endpoints (HTTP/ICMP) and exposes probe success/duration.
- **Why they matter:** They allow evaluation of both internal (API latency) and external/system-level health.

Grafana (visualization)

- **What it is:** Visualization & dashboarding for Prometheus and other sources.
- **Why Grafana:** Easy to build dashboards, add thresholds, and share screenshots for resumes/demos.

Alertmanager + alert bot

- **Alertmanager:** receives alerts from Prometheus and routes notifications (Slack/email/webhook).
- **Alert bot:** a small Python service listening to Alertmanager webhooks and performing automated remediation (restart container, scale worker, log incident). This demonstrates self-healing automation.

---

2. Why these particular choices (rationale)

- **Docker Compose** — easiest way to orchestrate multi-service demo on a dev machine. No K8s overhead.
- **Flask + Python** — your stack (you are comfortable with Python) and it's quick to instrument.
- **Postgres** — realistic DB for an L3-support / infra demo; demonstrates DB latency and query metrics.
- **Prometheus + Grafana** — industry-standard for metrics & dashboards; good resume value.
- **Exporters** — node_exporter and blackbox_exporter cover system-level and external probe monitoring without instrumenting OS code.
- **Alertmanager + webhook bot** — demonstrates a complete pipeline: metric → alert → human + automation.

---

3. What each container/service does in the demo (practical role)

1. **api (Flask)**
   - Serves /health and /transactions.
   - Exposes /metrics via prometheus-flask-exporter.
   - Demonstrates application-level request latency and errors.
2. **postgres_db (Postgres)**
   - Stores transactions table.
   - Worker inserts simulate production write load; API might read/write.
3. **worker**
   - Background job simulating asynchronous work (batch inserts, reports).
   - Gives realistic DB load and metrics (if instrumented).
4. **prometheus**
   - Scrapes metrics from api:8000/metrics, node_exporter, blackbox_exporter.
   - Evaluates alert rules (p95 latency, error rates).
   - Sends firing alerts to Alertmanager.
5. **node_exporter**
   - Exposes CPU, memory, disk, network usage from the host/container runtime.
6. **blackbox_exporter**
   - Probes api:/health and reports probe success/duration (helps detect endpoint reachability).
7. **grafana**
   - Visualizes Prometheus metrics in dashboards (API latency, errors, system metrics).
8. **alertmanager**
   - Receives alerts, groups them, routes notifications to Slack/email/webhook.
9. **alert_bot**
   - Receives webhooks from Alertmanager and runs remediation commands (e.g., docker restart flask_api), logs the incident.

---

4. How the demo simulates a production environment (design choices that mimic production)

- **Service separation:** API, DB, worker are separate containers (like microservices). This creates inter-service dependencies and network calls.
- **Persistent storage:** Postgres uses a Docker volume — simulates disk-backed DB.
- **Healthchecks & depends_on:** DB healthcheck and service dependencies simulate startup ordering; worker waits until DB is healthy.
- **Metrics & SLOs:** Exposing metrics (latency histograms, request counters) and creating SLO-like thresholds (p95 < 200ms, error rate <5%) simulates production monitoring targets.
- **External probing:** blackbox_exporter acts like external synthetic monitoring (real production often has external uptime monitors).
- **Automation/Remediation:** The alert bot demonstrates automated corrective steps (self-healing) — production systems may have automated playbooks or operators.
- **Test scenarios:** By injecting latency (sleep), killing containers, slowing DB, you can simulate major incidents (service outage, DB slowness, network partition) and validate monitoring + remediation.

---

5. Deep dive into metrics and instrumentation (practical how-to)

Metric types

- **Counter:** only increases (e.g., flask_http_request_total). Good for rates.
- **Gauge:** current value (e.g., process_resident_memory_bytes, or worker_queue_length).
- **Histogram:** buckets to measure latency distributions (e.g., flask_http_request_duration_seconds_bucket). Use histogram to compute quantiles with histogram_quantile().

Instrumentation best practices

- **Label cardinality:** avoid too many unique label values (e.g., don’t label by user_id). High cardinality kills Prometheus performance.

- **Use histogram for latency:** instrument the HTTP request latency as histogram (Flask exporter does this).
- **Counters for events:** increment on request completion, errors.
- **Expose /metrics:** Prometheus scrapes this endpoint.

Example PromQL explained

- rate(flask_http_request_total[1m]) → requests per second averaged over last minute.
- histogram_quantile(0.95, sum(rate(flask_http_request_duration_seconds_bucket[5m])) by (le)) → p95 latency calculated from histogram buckets over 5 minutes.
- rate(flask_http_request_exceptions_total[5m]) → error events per second.

---

6. Alerting design — rules, noise reduction, best practices

Rules we used (conceptual)

- **APIHighLatency**: p95 > 0.2s (200ms) for N minutes.
- **APIErrorRateHigh**: exceptions rate > threshold over 5 minutes.

Best practices

- **Use for** in alerts to avoid flapping (e.g., require condition for 1–5 minutes).
- **Group alerts** by alertname/service in Alertmanager to reduce noise.
- **Severity labels** (severity=warning vs severity=critical) to route differently.
- **Inhibition**: avoid duplicate alerts (e.g., if service is down, don't send DB-error alerts).
- **Silences** for maintenance windows.
- **Avoid low-level alerts for on-call** — escalate only meaningful incidents.
- **Alert fatigue:** refine thresholds and use recording rules for heavy queries.

---

7. Automation — safe self-healing patterns

- **What we implemented:** alert bot that restarts a container or scales worker on a defined alert.

- **Caution:** automated restarts can mask systemic issues — prefer:
  - *Auto-mitigation for transient faults* (e.g., auto-restart if crashloop detected).
  - *Human-in-the-loop for complex failures* (open ticket automatically but wait for confirmation before destructive actions).
- **Logging & audit:** bot writes to an incident_log.jsonl so each automated action is auditable.
- **Alternative production approaches:** use orchestration (Kubernetes) health checks + auto-restart policies, or implement operators/controllers rather than shelling out to docker socket.

---

8. How to simulate production incidents (practical tests)

1. **High latency:** modify /health to include time.sleep(1) → Prometheus sees p95 spike → alert fires.
2. **Errors:** throw an exception in /health or return 5xx.
3. **DB slowdown:** add time.sleep() around DB queries in API or reduce Postgres CPU (tc/netem for network).
4. **Kill a container:** docker stop flask_api → blackbox exporter and Prometheus detect downtime.
5. **Resource pressure:** run stress (or generate load) to spike CPU/memory and watch node_exporter metrics.
6. **Network partition:** (advanced) use tc or Docker network manipulation to drop packets.
7. **Verify automation:** when alert fires, watch alert_bot logs and docker ps to see container restarts or worker scaling.

---

9. Production hardening & what would change in real deployment

- **Never mount /var/run/docker.sock** into alert_bot for production — this gives full host control.
- **Secrets:** store credentials (Slack webhook, DB passwords) in secret stores (Vault, K8s Secrets); never commit .env.
- **Use orchestration:** Kubernetes with liveness/readiness probes, HPA for scaling; use controllers for remediation.
- **Use a metrics remote write/long-term storage** (Cortex, Thanos) for long retention and federation.
- **Add tracing & logs:** integrate OpenTelemetry for distributed tracing; add centralized log aggregation (ELK, Loki).
- **RBAC & network policies:** limit access to Grafana, Alertmanager; use HTTPS and auth (Grafana admin password, OAuth).

- **Alert throttling & escalation:** integrate with Ops tools (PagerDuty) for on-call management.

---

10. Repo readiness & what to include before pushing

- **Files to include:** all source, docker-compose.yml, prometheus/prometheus.yml, prometheus/alert.rules.yml, alertmanager/config.yml, Dockerfiles.
- **Files to exclude:** .env (use .env.example), local DB volumes, credentials.
- **README should include:**
  - Quick start
  - How to run tests (alert simulation commands)
  - Diagram and explanation of services
  - How to revert test changes (remove sleep)
- **.gitignore:** .env, db_data/, .DS_Store, __pycache__/
- **Add screenshots** for dashboards and an example alert firing (helps recruiters/readers).

---

11. Concrete practical checklist you can run now (final smoke tests)

1. docker compose up -d --build
2. docker ps → all services: api, db, worker, prometheus, grafana, node_exporter, blackbox_exporter, alertmanager, alert_bot.
3. Prometheus targets: http://localhost:9090/targets → ensure UP.
4. Grafana: http://localhost:3000 → add Prometheus data source http://prometheus:9090.
5. Generate traffic:

6. Check metrics: curl http://localhost:8000/metrics → you should see flask_http_request_total etc.
7. Trigger alert test (temporary change to app.py with time.sleep(1)), rebuild API: docker compose up -d --build api.
8. Wait 1–2 minutes → Alertmanager: http://localhost:9093 should show firing alert.
9. Check alert_bot logs: docker logs alert_bot → should have webhook payload and remediation actions.
10. Revert API temporary change.

---

12. Extensions & “next-day” deep dives we can do together

- PromQL deep-dive: recording rules, performance optimization.
- Add Postgres exporter and create DB-specific panels (query latency, lock counts).
- Add tracing (OpenTelemetry) and connect traces to metrics.
- Grafana provisioning (auto-import dashboards) and Alerting in Grafana.
- Move to Kubernetes (Helm charts) and replace alert bot with Kubernetes controller for safer remediation.
- Add anomaly detection (IsolationForest), and demonstrate automated threshold adjustments.

---

13. Final words — what to write in your README summary (short paragraph)

“**Observability Stack**: A containerized demo showcasing monitoring and automated remediation. The demo stacks a Flask API and Postgres DB with Prometheus and Grafana for metrics and dashboards, node/blackbox exporters for system and probe metrics, Alertmanager for routing alerts, and a lightweight Python alert-bot that demonstrates self-healing by restarting and scaling services. Use the repo to reproduce, test incidents, and learn practical observability and SRE techniques.”

6.