Project overview (quick-read)

Mini-K8s is a lightweight, single-binary container orchestrator you build from scratch. In spirit it re-creates the core control-plane behaviours of Kubernetes—declarative manifests, an idempotent reconcile loop, a scheduler, health-check–based restart logic, and basic observability—while staying small enough to finish in a few months. The end result runs on a laptop (or a couple of VMs), demonstrates real systems-engineering depth, and gives you a live demo that visibly reschedules pods when failures occur.

Implementation playbook (code-free version)

Phase 0 ‖ Groundwork (2 days)

Task	Outcome
Install Docker Engine / Docker Desktop.	`docker info` works.
Create a virtual environment; add Docker SDK, Pydantic, Typer, Prometheus-client, Pytest.	Clean runtime environment.
Run a one-liner using the Docker SDK to start and inspect a test container.	Verify SDK access.

Phase 1 ‖ Project scaffold (Week 1)

Task	Outcome
Initialise a Git repo; configure Black, Ruff, MyPy in *pre-commit*.	Consistent style & typing.
Create directories: `mini_k8s/`, `tests/`, `manifests/`, `docs/`.	Clear layout.
Add a Typer CLI with placeholder commands such as **apply** and **run**.	`$ mini-k8s --help` shows commands.
Set up a basic GitHub Actions workflow for lint + tests.	Green CI from day 1.

Phase 2 ‖ Models & manifest loader (Week 2)

Task	Outcome
Define Pydantic models for container specs, pod specs, restart policy, and pod status.	Strongly-typed objects.
Implement a loader that scans a manifest directory, validates YAML, and returns a list of pod specifications.	First CLI command prints parsed objects.
Write unit tests: malformed YAML → validation error; well-formed YAML → correct object structure.	Pass/fail harness in place.

Phase 3 ‖ Single-node launcher (Weeks 3-4)

Task	Outcome
Wrap Docker SDK in a small adapter that can create, start, stop, inspect, and list containers.	Clean abstraction layer.
Build a reconcile function that diffs *desired* manifests against *actual* running containers, producing a plan of create/stop operations.	Idempotent control loop.
Schedule the reconcile pass every few seconds inside an async task.	Pods start and stop predictably.
Provide unit tests that mock the adapter and assert diff correctness and idempotence.	Regression safety.

Checkpoint A — running `mini-k8s apply` starts one sample pod and never duplicates it on subsequent runs.

Phase 4 ‖ Node agent & health probes (Weeks 5-6)

Task	Outcome
Introduce a Node Agent process (can be a thread or separate process) that listens for operations via an in-memory or network queue.	Execution engine decoupled from controller.
Add a Health Manager that executes HTTP/TCP/command probes at specified intervals; on successive failures it enqueues a restart operation.	Automatic restarts mimic Kubernetes liveness checks.

Demonstrate by killing a container manually and watching the agent restart it according to the selected restart policy.	Failure-handling milestone.

Checkpoint B — container exits trigger automatic restart with a back-off timer.

Phase 5 ‖ Metrics & structured logging (Week 7)

Task	Outcome
Embed Prometheus metrics: counters for pods created/restarted, histogram for scheduling latency, gauge for running-pod count.	`/metrics` endpoint exports data.
Switch to JSON logs using a structured logger; include pod name, node name, event type.	Greppable, machine-readable logs.
Optional: load a Grafana dashboard JSON showing key metrics.	Observability proof.

Phase 6 ‖ Multi-node scheduling (Weeks 8-9)

Task	Outcome
Spin up a second Docker host (remote daemon or VM); expose it over TCP with TLS disabled for local testing.	Multi-host playground.
Create a Node abstraction that tracks capacity, labels, and heartbeat.	Scheduler input.
Implement a simple scheduling algorithm (round-robin, then least-loaded).	Pods land on different nodes.
Send periodic heartbeats from Node Agents; mark a node *Unknown* if heartbeats cease and reschedule its pods.	Basic high availability.

Checkpoint C — killing Node A causes its pods to reappear on Node B within a set timeout.

Phase 7 ‖ Advanced features (Weeks 10-12)

Feature	Goal

Rolling updates	Define a higher-level *PodSet* spec with replicas and strategy; implement create-new / wait-ready / delete-old flow.
Horizontal Pod Autoscaler	Separate controller polls Prometheus CPU metric; adjusts replicas up/down toward a target.
Taints & tolerations	Scheduler skips nodes with unsatisfied taints, enabling node-class isolation.

Phase 8 ‖ High-availability control-plane (Weeks 13-14)

Task	Outcome
Persist cluster state to a small Raft log using an off-the-shelf async library.	Survives controller restarts.
Run three controller instances; only the Raft leader mutates state.	No single point of failure.
Add a readiness endpoint that reveals leadership status for monitoring.	Ops visibility.
Chaos test that terminates the leader and measures recovery time.	Reliability evidence.

Phase 9 ‖ Polish & release (Weeks 15-16)

Task	Outcome
Compose a one-command demo via Docker Compose that starts two nodes, a controller, and Grafana.	Easy trial for reviewers.
Record a short screencast: apply manifest → pods run → kill node → pods reschedule and graphs update.	Visual proof of functionality.
Write an architecture document with a diagram, API endpoint table, and sequence diagrams for reconcile and scheduling flows.	Demonstrates communication skills.
Publish to PyPI (or Homebrew) and tag release v0.1.	Public distribution.
Blog post titled “Building a Kubernetes-style orchestrator in N lines of Python.”	Marketing piece for recruiters.

Key checkpoints recap

Checkpoint	Visible behaviour
A	Single manifest starts container, reconcile loop idempotent.
B	Health probes trigger restart policy.
C	Multi-node failover: pods automatically reschedule when a node disappears.
Final	Full demo with metrics, dashboards, and chaos test passes in CI.

Study resources (all code-free)

1. **Docker SDK for Python documentation** — read the lifecycle and events sections.
2. Official **Kubernetes Basics** tutorial — skim to internalise concepts like desired state and controllers.
3. Brendan Burns' *Designing Distributed Systems* — chapters on controllers and scheduling patterns.
4. Liz Rice's "Containers from Scratch" talk — understand namespaces and cgroups conceptually.
5. Prometheus Python client README — focus on metric types and naming conventions.

Tips for staying on track

- **Time-box** deep dives; file tickets for enhancements instead of stalling MVP delivery.
- **Interface-first** design: write docstrings and method signatures before implementation.
- **Automate tests and linting early** so new features don't regress earlier phases.
- **Document decisions** (e.g., why a particular scheduling heuristic) right in the repo.

Complete the list through Phase 6 and you already have a résumé-ready demo; reach Phase 8 and you'll possess a standout systems project few new grads can match.