Project Management — Koorde vs Chord Benchmark

1. Project Overview & Inspiration

This project evaluates the performance of distributed hash tables (DHTs) in a realistic cloud environment. It was inspired by:

1. **The Koorde Paper (Kaashoek & Karger, 2003):** Proposed a degree-optimal DHT with $O(\log N / \log \log N)$ diameter.
2. **Existing Benchmarks:** Online comparisons of Chord vs. Kademlia, (notably https://github.com/macvincent/slbdch), which we aimed to extend by validating Koorde's theoretical properties.

Goal: Move beyond simple algorithmic simulation to a full systems engineering evaluation, measuring **latency**, **cache hit rate**, and **throughput** in a containerized, orchestrated environment.

2. Team Roles & Responsibilities

To manage the complexity of a distributed systems project, we divided responsibilities based on the engineering stack layers.

Role	Member	Key Responsibilities
Protocol Engineer	Yuzheng Shi	**Core Logic & Correctness:** • Implemented Koorde routing (de Bruijn graph) & Chord finger tables. • Validated routing correctness and stabilization logic. • **Experiments:** Conducted Latency and Cache Hit Rate (Churn) experiments.
Infrastructure Lead	Anran Lyu	**Deployment & Scale:** • Containerized the application (Docker) and designed K8s manifests. • Managed AWS EKS infrastructure and LocalStack testing. • **Experiments:** Led Throughput/Saturation testing and distributed Locust load generation.

3. Project Timeline & Milestones

The project followed an iterative engineering lifecycle over the course of the term.

Phase	Milestone	Description	Status
Phase 1	**Core Implementation**	Implemented `DHTNode` interface, Koorde logic (`logicnode`), and Chord logic (`chord`). Verified routing locally.	✅ Completed
Phase 2	**Local Integration**	Built Docker images (`node.Dockerfile`) and created `docker-compose` / LocalStack setups to test multi-node interaction.	✅ Completed
Phase 3	**Cloud Deployment**	Migrated to AWS EKS. Solved IAM and quota issues. Established a stable 8-32 node cluster environment.	✅ Completed
Phase 4	**Benchmarking**	Executed 3 major experiments: Latency Scaling, Churn Resilience, and Throughput Saturation.	✅ Completed
Phase 5	**Analysis & Reporting**	Aggregated CSV results, generated Python plots, and produced the final `experiemnt_report.md`.	✅ Completed

4. Architecture & Implementation Breakdown

We structured the repository to separate concerns, allowing parallel development.

A. Routing Layer (The "Brain")

Owned by Protocol Engineer

- **Koorde:** Implemented in `internal/node/logicnode` and `internal/node/routingtable`. Challenges included managing "imaginary" nodes and de Bruijn graph transitions.
- **Chord:** Implemented in `internal/node/chord`. Focused on standard stabilization and finger table maintenance.

B. Infrastructure Layer (The "Body")

Owned by Infrastructure Lead

- **Containerization:** `docker/` contains optimized builds for the Node and Client.
- **Orchestration:** `deploy/eks` contains the Kubernetes manifests (Deployments, Services, ConfigMaps) required to run the cluster in AWS.
- **Automation:** `scripts/` contains PowerShell automation (e.g., `test-koorde-scaling.ps1`) to orchestrate complex test scenarios.

C. Shared Components

Jointly Developed

- **Web Cache:** `internal/node/cache` implements the LRU cache and hotspot detection.
- **RPC Layer:** gRPC definitions in `proto/` ensure consistent communication between nodes.

5. Risk Management & Adaptations

We encountered significant hurdles that required agile adaptation of our project scope.

Risk / Problem	Impact	Mitigation / Adaptation
AWS Learner Lab Quotas	Hard limit of ~9 EC2 instances prevented planned 100-node physical cluster.	**Scope Adjustment:** Refocused on detailed analysis of small-to-medium clusters (8-32 nodes). Proposed logical scaling and simulation for asymptotic validation.
Sandbox IAM Restrictions	AWS Student Sandbox lacked IAM roles for EKS, blocking initial deployment.	**Platform Migration:** Reverted to AWS Learner Lab and optimized resource usage (packing multiple pods per node) to maximize cluster size within limits.
Load Generator Crashing	Local machine CPU exhausted when simulating >1000 users with Locust.	**Distributed Testing:** Architected a distributed Locust setup on AWS (1 Master + N Workers) to generate stable load up to 4000 users.

6. Key Artifacts & Evidence

The following artifacts in the repository demonstrate the completion of our goals:

- **Source Code:**
  - Koorde Logic: `internal/node/logicnode/`
  - Chord Logic: `internal/node/chord/`
- **Infrastructure:**
  - K8s Manifests: `deploy/eks/`
  - Dockerfiles: `docker/`
- **Data & Analysis:**
  - Benchmark Report: `BENCHMARK_REPORT.md`
  - Raw Data: `test/results/32nodes/*.csv`
  - Visualization Script: `test/generate_plots.py`

7. Final Reflection

What began as a theoretical implementation exercise evolved into a comprehensive distributed systems engineering project. We successfully:

- Built a functional, multi-protocol distributed cache.
- Navigated real-world cloud constraints (quotas, IAM).
- Produced empirical data comparing theoretical promises (Koorde) vs. practical reality (Chord).

While we could not demonstrate Koorde's asymptotic superiority due to scale limits, we successfully proved that **Chord is more stable and performant** for small-to-medium clusters, a valuable negative result for system architects.