Cloud Administration
Semester Project

Daniel Wladdimiro
ESILV
daniel.wladdimiro@devinci.fr

Second semestre 2024-2025

1 Semester Project

1.1 Project Spirit

This is an open-ended engineering project. There is no imposed architecture and no imposed technology stack. Your team proposes a solution, justifies design decisions, and demonstrates performance improvement with evidence.

The theme is intentionally simple:

- you select a real dataset stored in a database,
- your system exposes queries under concurrent load,
- you iterate on the design to improve query performance (e.g., scaling, caching, asynchronous processing, etc.).

1.2 Motivation

Database query performance is a frequent bottleneck in cloud applications. Performance often degrades due to:

- inefficient queries (large scans, poor filtering, expensive joins),
- limited concurrency (locks, connection limits, contention),
- queueing effects that amplify tail latency (p95/p99),
- bottlenecks moving between layers when the system scales,
- "quick fixes" introducing new risks (stale caches, retry storms, cost growth).

The objective is to learn how to reason about these constraints, choose a coherent architecture for a workload, and defend trade-offs.

1.3 Teams and Final Session

- Teams of **3 students**.
- Final evaluation happens during the **last TD**: **15 minutes per team**.
- Format is free: live demo, slides, whiteboard explanation, or a combination.
- Deployment choices are free: local machines, personal VMs, or any cloud provider. Using **GCP is optional**.

1.4 Challenge Statement

Design a small data service that answers queries over a dataset. Then improve performance under load, and explain:

- what the initial bottlenecks were (hypotheses + observations),
- what you changed (and why),
- what measurable impact you obtained,
- what trade-offs you accepted (complexity, consistency, operational effort, cost).

1.5 Design Freedom

You choose:

- the database technology (SQL or NoSQL),
- the API style (REST, gRPC, GraphQL, etc.),
- whether and where to use caching,
- whether and how to use a broker (or any asynchronous mechanism),
- whether to scale horizontally, vertically, or both,
- how many components you create (monolith or multi-service),
- where you deploy (local / VMs / cloud).

If you want the project to reflect distributed-systems ideas clearly, it is generally helpful to demonstrate multiple running components/instances (e.g., several nodes or replicas). How you do this is your decision.

1.6 Dataset: Where to find data

Use a real dataset (CSV/JSON) that supports meaningful queries (filters, aggregations, top-N). Recommended sources:

- Kaggle Datasets: https://www.kaggle.com/datasets
- UCI Machine Learning Repository: https://archive.ics.uci.edu/
- Google BigQuery Public Datasets: https://cloud.google.com/bigquery/public-data
- AWS Registry of Open Data: https://registry.opendata.aws/

1.7 Suggested directions (examples, not instructions)

You may explore, if relevant to your design:

- **Database-side techniques:** indexes, query rewriting, pagination strategies, schema adjustments, connection pooling.
- **Caching:** cache-aside, TTL decisions, invalidation approach, stampede mitigation, cached aggregates.
- **Scaling:** replicate stateless services, isolate heavy endpoints, tune concurrency limits, reduce contention.
- **Asynchronous design / broker:** background aggregation jobs, cache warmers, async heavy queries, ingestion events.
- **Observability:** throughput, error rate, p95/p99 latency, saturation signals, simple dashboards.
- **Trade-offs:** consistency vs performance, simplicity vs features, cost vs throughput.

1.8 What to show in the final 15-minute session

There is no written report. The goal is to convince an engineering audience.

1) System explanation (one diagram)

Present your architecture and how requests/data flow through the system.

2) Evidence of improvement

Show a clear before/after story. Typical evidence includes:

- throughput (requests/sec or queries/sec),
- latency percentiles (p50/p95/p99),
- error rate/timeouts,
- at least one supporting signal (e.g., query plan comparison, cache hit rate, connection count, CPU saturation, queue length).

Any load-testing tool is acceptable (k6, Locust, JMeter, etc.). What matters is clarity and credibility of results.

3) Decision justification

For each major decision, explain:

- **why** you chose it,
- **what problem** it addressed,
- **what trade-off** it introduced.

1.9 Optional GCP mapping

If you choose GCP, you can keep it simple (illustrative examples):

- Compute Engine for manual multi-instance deployment,
- Cloud Run or GKE for horizontal scaling of an API,
- Cloud SQL for managed PostgreSQL/MySQL,
- Memorystore for Redis caching,
- Pub/Sub as a managed broker.

1.10 Scope control

To keep the project manageable:

- keep the API small (a few query endpoints are enough),
- focus on a small set of query workloads and optimize them well,
- prioritize clarity, evidence, and reasoning over adding many technologies,
- avoid building a full UI (optional).

2 Evaluation Rubric (Final TD) — /20

The evaluation rewards **sound engineering reasoning**, **measurable evidence**, and **clear justification of trade-offs**. This rubric describes *how* the project will be assessed; it does not prescribe *what* you must build.

Evaluation		
Category	**Criteria**	**Points**
Problem Framing & Dataset	Problem relevance: the team clearly defines the use case and explains why query performance is a meaningful engineering challenge in this context.	2
	Dataset fit: the dataset choice supports meaningful query workloads and the project goals (size/structure/fields aligned with the scenario).	2
	Workload definition: the team defines representative query patterns and explains why these queries matter (filters, aggregations, top-N, etc.).	2
System Design & Trade-offs	Architecture clarity: components, data flow, and responsibilities are clearly explained (diagram quality matters).	3
	Decision justification: major choices are defended with explicit trade-offs (performance, complexity, correctness/consistency, cost, operational effort).	3
	Robustness thinking: the team identifies key risks for their design (contention, overload, staleness, retries, hotspots) and explains how they are handled at an appropriate level.	2
Performance Evidence & Analysis	Baseline vs improved: the team presents a clear before/after comparison with consistent conditions and a coherent improvement narrative.	3
	Metrics quality: the team reports meaningful metrics (throughput, p95/p99 latency, error rate) and links results to system behavior.	3
	Explanation quality: the team connects changes to observed effects and explains remaining bottlenecks/limits.	2
Implementation & Reproducibility	Technical quality: the system runs as presented; code/configuration are readable and coherent.	2
	Reproducibility: the team can explain how to run the system and reproduce the demo/test quickly (setup steps, configuration, dataset loading).	1
Oral Defense (15 min)	Clarity and structure: the presentation is well organized, fits within time, and is easy to follow.	1
	Q&A: the team answers questions accurately and defends choices with technical arguments.	1
	Total	**20**