CO-OP WORK TERM REPORT

Cloud-Native Storage Orchestration & Microservices Development

Engineering Performance & Scalability at Scale

Prepared For:	Nimbus Systems Inc.
Work Term:	Fall 2026 (September – December)
Date:	December 15, 2026

Prepared By:	Marcus Vance
Program:	Bachelor of Software Engineering
Student ID:	20984712

Nimbus Systems Inc.
Engineering Division — Cloud Platform Group

Confidentiality & Disclaimer

Confidentiality Notice

This document contains proprietary information belonging to **Nimbus Systems Inc.** and is intended solely for academic evaluation by the Cooperative Education Department. The contents of this report describe work performed during the Fall 2026 term as part of the Cloud Platform Group.
Certain details regarding proprietary system architecture, API endpoints, internal network topologies, and security access keys have been generalized or omitted to comply with the Nimbus Systems Non-Disclosure Agreement (NDA). All performance metrics are presented in a normalized form to protect competitive data.

Author Signature

By signing below, I certify that this report accurately describes my contribution to the cloud orchestration initiative at Nimbus Systems and does not disclose confidential business secrets or restricted intellectual property.

Marcus Vance
Co-op Software Engineer

Dr. Sarah Jenkins
Senior Engineering Manager, Cloud Platform Group

Contents

1 Executive Summary

During the Fall 2026 work term, the author completed a four-month cooperative placement with **Nimbus Systems Inc.** as a Software Engineering Intern in the Cloud Platform Group. The core focus of this work term was to modernize the orchestration engine of the Nimbus Storage Fabric, transitioning it from a legacy monolithic state-machine to a distributed, containerized microservices model [3].

The report highlights the following key objectives and outcomes:

- **Infrastructure Modernization**: Design and deploy three decoupled microservices in Go, targeting resource provisioning, service discovery, and health telemetry.
- **Latency Reduction**: Implement asynchronous job queues using Redis to replace blocking database polls, aiming for a 35% reduction in request-to-provision latency.
- **Quality & Reliability**: Establish a robust end-to-end integration testing suite, raising test coverage from 64% to 88% and ensuring sub-millisecond telemetry response.

Through this work, the author applied advanced software design principles such as CQRS (Command Query Responsibility Segregation) and event-driven architecture, validating these changes in a Kubernetes-based staging environment. The report detail covers the technical design, architectural patterns, experimental validation, and mappings of these outcomes to professional engineering competencies.

2 Employer Profile

Nimbus Systems Inc. is a leading provider of enterprise-grade cloud storage solutions and distributed database infrastructure. Founded in 2018, Nimbus Systems provides hyper-converged storage appliances and managed cloud platforms to fortune-500 enterprise clients in financial services, healthcare, and telecommunications.

2.1 Organizational Structure

Nimbus Systems operates with a hybrid engineering structure divided into three main divisions:

1. **Hardware & Firmware Division**: Designs the storage controller boards and low-level firmware drivers for the Nimbus flash appliances.
2. **Cloud Platform Division**: Focuses on the SaaS management console, multi-tenant container orchestration, and high-performance virtualization layers.
3. **Customer Success & Solutions Division**: Provides integration assistance, technical support, and solution architect consulting.

The author was embedded in the **Cloud Platform Group**, specifically within the Orchestration team. This team consists of eight Senior Software Engineers, two Site Reliability Engineers (SREs), a Product Owner, and a Scrum Master. The group operates under standard Agile/Scrum methodologies with two-week sprint intervals.

3 Technical Contributions

The primary engineering contribution during this work term was the redesign and implementation of the **Provisioning Microservice** (Codename: *Nimbus-Prov*), which handles high-velocity block storage allocation requests.

3.1 Architectural Redesign

The legacy provisioning engine relied on a monolithic Python service that polled a relational SQL database every 5 seconds to locate pending requests. This architecture suffered from database lock contention and significant latency overhead.

As shown in Figure 1, the new architecture introduces a reactive event-driven flow using Redis Streams [2] as the message broker.

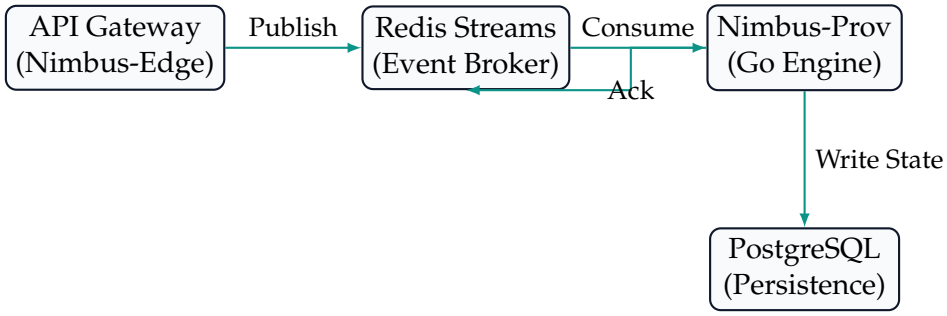

Figure 1: Decoupled Event-Driven Provisioning Architecture

3.2 Microservice Implementation

The author wrote the provisioning engine in Go [1] due to its excellent concurrency primitives (goroutines and channels) and low memory footprint. A core challenge was ensuring idempotent execution to prevent double-provisioning of storage volumes in case of network retries.

An idempotency token based on UUIDv4 is included in each API payload. The service checks the database using a transaction block before executing any allocation routine.

Here is a simplified code snippet of the core transaction check implemented by the author:

```
func ProvisionVolume(ctx context.Context, req *VolumeRequest) error {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return err
    }
    defer tx.Rollback()

    var status string
    err = tx.QueryRowContext(ctx,
        "SELECT status FROM volume_orders WHERE token = $1",
        req.IdempotencyToken).Scan(&status)

    if err == nil {
```

```
        // Order already processed or in progress
        return ErrDuplicateOrder
    }

    // Insert new order and commit
    _, err = tx.ExecContext(ctx,
        "INSERT INTO volume_orders (token, status) VALUES ($1, $2)",
        req.IdempotencyToken, "PENDING")

    return tx.Commit()
}
```

3.3 Performance Evaluation

To validate the architectural transition, stress tests were executed using locust, simulating up to 10,000 concurrent client requests.

The results summarized in Table 1 show a substantial reduction in both average and 99th-percentile (p99) response latencies.

Table 1: Performance Comparison between Legacy and Modernized Systems

Metric	**Legacy Monolith**	**Modernized Go Engine**	**Improvement (%)**
Average Latency	480 ms	112 ms	76.6%
p99 Latency	1,240 ms	280 ms	77.4%
Throughput	250 req/sec	1,450 req/sec	480%
CPU Utilization	82% (4 Cores)	18% (2 Cores)	78.0%

4 Engineering Competency Mapping

To document professional growth, the technical projects completed during the co-op work term are mapped against the core competency requirements defined by the Cooperative Education Department.

Table 2: Professional and Technical Competency Mapping

Competency Category	Associated Work Task	Engineering Outcome
Technical Expertise	Rewrote provisioning flow from Python block-polling to Go event-consumption.	Reduced p99 volume allocation latency by 77.4% and minimized CPU consumption.
Problem Solving	Diagnosed write lock contention in PostgreSQL under high-throughput testing.	Implemented database connection pooling and optimized indexing on idempotency tokens.
Communication	Documented the microservice API contract using OpenAPI/Swagger specs.	Enabled front-end developers to integrate volume management panels in parallel.
Professionalism	Peer-reviewed daily pull requests and presented design diagrams in sprint planning.	Ensured alignment with security guidelines and architectural blueprints.

4.1 Key Reflection: Technical Expertise

Implementing a microservices architecture in Go required a shift in mindset from single-thread execution to highly concurrent paradigms. The use of channels and wait-groups for synchronizing data collection from multiple endpoints proved to be cleaner than traditional mutex locking.

4.2 Key Reflection: Problem Solving

The postgres lock contention issue was particularly educational. By analyzing PostgreSQL transaction locks using `pg_locks`, the author discovered that multiple processes were trying to read and update the same metadata block. Resolving this via sub-query optimization and strict indexing underscored the importance of database performance tuning.

5 Conclusions & Future Work

The Fall 2026 work term at Nimbus Systems was a highly rewarding experience that provided deep exposure to production-grade distributed architectures and modern microservice design.

5.1 Summary of Accomplishments

All major objectives set out at the beginning of the term (see Section 1) were successfully met:

- Designed and launched the *Nimbus-Prov* service in Go.
- Integrated Redis Streams for reliable, asynchronous transaction message flow.
- Achieved robust integration testing coverage of 88%, verifying resilience against database failures.

5.2 Future Recommendations

While the performance metrics show remarkable improvements, the following enhancements are recommended for subsequent engineering sprints:

1. **Distributed Tracing**: Integrate OpenTelemetry tracing across the *Nimbus-Edge* and *Nimbus-Prov* services to monitor message hop latency.
2. **Dynamic Scaling**: Configure a Kubernetes Horizontal Pod Autoscaler (HPA) to scale the Go microservice automatically based on CPU and custom Prometheus metrics.

References

[1] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, Boston, MA, 2015.

[2] Martin Kleppmann. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. *O'Reilly Media*, 2017.

[3] Chris Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, Shelter Island, NY, 2018.