Week 1 — Cloud Computing as a Systems Discipline

Foundations: Scale, Failure, Declarative Control

Dr. J. Burns

INFT 3611

November 29, 2025

Section 1: Introduction

This content is based on the following public resources: https://www.distributed-systems.net/index.php/books/ds4/

Cloud Computing as a Systems Discipline

Core Perspective

Cloud computing is fundamentally a **distributed systems discipline**, where correctness depends on algorithmic coordination across unreliable and asynchronous components.

- Cloud platforms behave more like planet-scale operating systems than virtualized datacenters [1].
- Core constraints include:
  - absence of globally consistent state,
  - nondeterministic event ordering,
  - continuous node and process churn.
- These conditions require distributed algorithms that ensure convergence despite message delay and partial observability.

Why Cloud Requires Systems Thinking

Misconception

"Cloud computing is just servers in someone else's datacenter."

- Large production systems (Google, AWS, Azure) operate under **persistent instability**: crashes, partitions, and restarts occur continuously [2].
- Procedural logic assumes:
  - linear execution,
  - reliable infrastructure,
  - timely intervention.

  These assumptions fail at scale.
- Correctness requires:
  - idempotent operations,
  - eventual consistency,
  - monotonic state evolution [3].

Cloud as a Large-Scale Operating System

Analogy

Cloud control planes approximate operating system kernels, but operate across unreliable devices, asynchronous communication, and weak consistency.

- Responsibilities include:
  - global workload scheduling [4],
  - policy governance,
  - identity and isolation,
  - reconciliation of desired and observed state.
- Unlike local kernels, cloud systems cannot rely on:
  - atomic memory,
  - global monotonic time,
  - reliable message delivery.
- Correctness emerges from distributed convergence, not sequential execution.

Design Tensions in Cloud Architecture

Structural Tensions

Global-scale systems must balance fundamental constraints that cannot be optimized simultaneously.

- **Consistency vs throughput:** strong consistency requires coordination, limiting parallelism [5].
- **Elasticity vs predictability:** scaling shifts resource topology unpredictably.
- **Automation vs visibility:** automation improves correctness but reduces operator observability.
- **Availability vs correctness:** high availability often permits temporary inconsistency.

Scale Amplifies Complexity

Scaling Reality

At cloud scale, rare events become constant. Small failure probabilities multiplied across millions of components produce continuous disturbances.

- Industry data shows that large fleets experience ongoing hardware and software faults [6].
- The system behaves like a high-dimensional dynamical system under continuous perturbation.
- Manual reasoning cannot track or correct drift at this speed.
- Stability requires automated control loops that re-establish invariants despite ongoing disturbances.

Continuous Perturbation as the Normal State

Operational Reality

At large scale, the system is **never quiescent**. Failures, recoveries, and timing anomalies form a continuous background process.

- Node churn (joins, leaves, crashes) occurs constantly in large fleets [2].
- Network delay, jitter, and loss vary dynamically across datacenter fabrics.
- Clock skew accumulates between nodes, undermining assumptions about event ordering [3].
- Replicated components diverge due to nondeterministic propagation delays.
- Systems must rely on convergence, not instantaneous correctness.

The State Explosion Problem

Key Idea

Cloud systems maintain multiple interacting state domains whose cross-product produces an intractably large global state space.

- State classes include:
  - configuration state,
  - runtime and health state,
  - dependency and topology state,
  - policy and security state.
- Even small inconsistencies can cascade across these interconnected layers.
- Strongly consistent global snapshots require coordination mechanisms such as consensus [3], which becomes expensive at scale.
- Automated controllers are required to continually manage drift within this state space.

Limits of Manual Coordination

Human Limitation

Human operators cannot observe, reason about, or repair distributed state quickly enough to maintain correctness in highly concurrent environments.

- Manual intervention introduces variable latency, allowing inconsistencies to widen.
- Operators often apply partial or inconsistent fixes, unintentionally increasing drift.
- Post-incident reports frequently attribute outages to configuration errors and slow reaction paths [6].
- The rate of topological change exceeds human cognitive limits.

Why Automation Becomes Mandatory

Benefits of Automated Controllers

Automation enforces system invariants at speeds and levels of consistency impossible for human operators.

- Automated controllers:
    - detect drift immediately,
    - execute corrective actions uniformly,
    - operate continuously rather than episodically.
- Organizations such as Google and AWS rely on continuous automated monitoring and reconciliation [6].
- Automation transforms correctness from manual labor into a property maintained by algorithmic control.

Coordination as a Control Problem

Control-Theoretic Interpretation

Distributed systems resemble feedback control systems, where noisy observations and delayed actions must still lead to stable behavior.

- The core loop: **Observe** $\rightarrow$ **Compare** $\rightarrow$ **Correct**.
- Controllers must maintain stability under:
  - partial observability,
  - asynchronous updates,
  - unpredictable disturbances,
  - variable execution latency.
- The primary correctness property is **convergence**, not perfect instantaneous agreement.
- This insight underpins modern control-plane architectures used in Kubernetes, Borg, and distributed service fabrics.

Failure as a First-Class Property

Principle

In cloud environments, **failure is a continuous operating condition**, not an exceptional event.

- Large-scale systems observe constant hardware faults, process exits, link failures, and retries [2].
- Components fail independently, producing diverse and unpredictable patterns.
- Software must assume that any communication, computation, or storage operation may not complete as expected.
- Reliability emerges from redundancy, feedback loops, and corrective algorithms—never from individual component durability.

Partial Observability

Fundamental Constraint

Distributed systems never possess a fully accurate, real-time global view of state.

- Observations are:
  - delayed,
  - partial,
  - stale,
  - inconsistent across replicas.
- Controllers must take action despite incomplete knowledge.
- Event timestamps cannot be fully trusted due to clock skew and clock drift [3].
- Systems must be engineered to remain stable under uncertainty.

Resilient System Behavior

Common Resilience Techniques

Distributed systems use operational properties that remain correct under retries, reordering, and partial execution.

- **Idempotent operations**: issuing the same update multiple times produces the same final state.
- **Monotonic state transitions**: state evolves in one direction, tolerating out-of-order messages.
- **Eventual convergence**: all replicas converge on a stable state after disturbances.
- These principles strengthen correctness in the presence of nondeterministic failures and message delivery conditions.

Why Declarative Instead of Procedural

Issue

Procedural instructions rely on ordering guarantees that distributed systems cannot provide.

- A sequence of procedural steps will fail if:
  - messages are delayed,
  - node scheduling varies,
  - operations are retried,
  - partial failures interrupt execution.
- Declarative intent defines the target invariant; controllers compute the needed transitions.
- This shifts correctness from step-by-step execution toward long-run convergence [3].

Intent as System Specification

Definition

Declarative intent specifies the desired stable properties of the system rather than the sequence of steps to achieve them.

- Operators express targets such as:
    - replica counts,
    - topology constraints,
    - policy and security requirements,
    - resource limits.
- Controllers continuously reconcile observed state with this specification.
- Self-healing behavior emerges because the intent remains stable even as the environment changes.

Intent Drift

Sources of Drift

Differences between an operator's declared intent and the system's actual condition accumulate continuously in distributed environments.

- Causes of drift:
  - node failures and evictions,
  - message delays and reordering,
  - resource contention,
  - propagation latency between replicas.
- Drift is unavoidable; systems must detect and correct it rather than assume stable state.
- Persistent automated reconciliation is required to maintain correctness at scale.

Desired State

Definition

The **desired state** is the operator-defined specification of how the system should behave under correct conditions.

- Examples:
  - “Maintain 5 replicas of service X.”
  - “Ensure all nodes enforce network policy Y.”
  - “Keep workloads within zone and quota constraints.”
- The desired state is stable, declarative, and often version-controlled.
- It acts as the reference point for control-plane decisions.

Observed State

Reality

The **observed state** reflects the system's actual behavior but is always partial, stale, and approximate in distributed environments.

- Observed state is influenced by:
  - network delays,
  - local caching,
  - clock skew,
  - inconsistent updates across replicas.
- No controller sees the whole cluster instantaneously.
- Decision making must tolerate incomplete, outdated, or ambiguous observations [3].

State Comparison

Mechanism

Control planes continuously compute corrective actions based on the difference:

$$\text{Correction} = \text{Desired State} - \text{Observed State}$$

- Controllers identify:
  - missing resources (under-replication),
  - excess resources (over-provisioning),
  - inconsistent policy or configuration.
- This differential approach is robust to reordering and partial failure.
- Reconciliation logic is idempotent and monotonic, enabling safe repeated execution.

Reconciliation Loop

Control Loop Model

The canonical control-plane loop:

$$\textbf{Observe} \rightarrow \textbf{Compare} \rightarrow \textbf{Correct}$$

drives convergence toward the declared intent.

- Controllers re-evaluate state continuously, not on demand.
- Reconciliation absorbs disturbances caused by failures, delays, and nondeterministic event ordering.
- Long-term correctness is achieved through **eventual convergence**, not perfect instantaneous agreement [6].
- This strategy underlies modern systems such as Kubernetes, Borg, and Mesos.

Idempotence and Safety

Key Principle

Corrective operations must be **idempotent**: safe to repeat, reorder, or retry without producing unintended side effects.

- Idempotence protects correctness when:
  - messages are duplicated,
  - operations are retried,
  - partial failures interrupt updates,
  - controllers race to modify shared state.
- Modern control-plane implementations, including Borg and Kubernetes, rely heavily on idempotent handlers for stable reconciliation [4].
- Idempotence forms the "safety net" for distributed convergence.

Distributed Reconciliation Dynamics

Challenges in Multi-Controller Systems

Multiple controllers operate concurrently and may interact in unexpected ways.

- Risks include:
  - oscillation (controllers undo each other's changes),
  - race conditions in shared state,
  - inconsistent enforcement of cross-cutting policies.
- Stability requires:
  - clearly defined ownership of state domains,
  - deterministic conflict-resolution strategies,
  - eventually consistent propagation of updates.
- Formal reasoning increasingly informs the design of modern controllers [5].

Case Study: Replica Restoration

Scenario

A replicated service must maintain the invariant: **exactly N healthy replicas** should exist.

- Failure events remove replicas (crashes, eviction, unreachable nodes).
- The observed state becomes: actual replicas $< N$.
- Controllers perform:
  - detection of missing replicas,
  - scheduling or placement decisions,
  - instantiation of new replicas.
- Even under partial failure, the system converges back to the declared invariant through automated reconciliation.

Case Study: Policy Enforcement Under Failure

Scenario

A network or security policy must be enforced globally, but partial failures lead to inconsistent rule deployment.

- Individual nodes may:
  - fail to apply required rules,
  - apply outdated or conflicting rules,
  - lose local state due to restart or eviction.
- Controllers detect drift via periodic inspection and update propagation.
- Convergence resembles epidemic or anti-entropy protocols used in distributed databases [5].
- The system gradually restores global policy consistency without manual intervention.

Case Study: Elastic Scaling

Scenario

Workload load increases or decreases, triggering an elasticity policy governed by declarative constraints.

- The desired state encodes elasticity bounds (min/max replicas, scaling rules).
- The observed state reflects resource saturation or underutilization.
- Controllers:
  - resize replica sets,
  - redistribute load,
  - rebalance across zones or failure domains.
- Elastic scaling loops must avoid oscillation and ensure stable response dynamics [6].

Self-Stabilizing Control Planes

Insight

Modern cloud control planes approximate **self-stabilizing distributed systems** that converge toward correctness from arbitrary states.

- The concept originates from Dijkstra's theory of self-stabilization.
- Control planes repeatedly correct drift through reconciliation loops.
- Convergence occurs despite:
  - stale data,
  - transient inconsistency,
  - partial failures,
  - message reordering.
- This explains why declarative cloud systems recover automatically from widespread disturbances.

Why Declarative Control Dominates

Architectural Reasoning

Declarative control is the only scalable strategy for correctness in asynchronous, failure-prone environments.

- Step-by-step procedural logic breaks under concurrency, races, and retries.
- Declarative intent provides a stable “truth anchor” for reconciliation.
- Long-run correctness derives from the convergence of multiple controllers acting independently.
- This pattern underlies the design of Kubernetes, Borg, Mesos, and large cloud fabrics [4].

Summary of Week 1

Key Takeaways

- Cloud computing must be analyzed as a distributed systems discipline.
- Scale introduces continuous failure, partial observability, and nondeterminism.
- Declarative intent defines system invariants; observed state reflects imperfect reality.
- Reconciliation (Observe $\rightarrow$ Compare $\rightarrow$ Correct) is the core mechanism for achieving convergence.
- Idempotence, monotonicity, and self-stabilization ensure robustness under reordering and retries.

Suggested Activities

Activities for Students

- Analyze a distributed system (e.g., Cassandra, etcd) and identify how it handles drift.
- Explain how idempotence prevents race conditions and supports retries.
- Compare reconciliation-based coordination vs consensus-based coordination.
- Review an incident report and identify how declarative or procedural control contributed to system stability or instability.

End of Week 1

Questions?

Discussion?

Preparation for Week 2: Models of Distributed Computation

References I

- A. Verma, L. Cherkasova, R. Campbell, and G. Shen, "Large-scale cluster management at Google," *Proceedings of the 2008 International Conference on Autonomic Computing*, 2008.
- J. Hamilton, "On Designing and Deploying Internet-Scale Services," *Proceedings of the Large Installation System Administration Conference (LISA)*, 2007.
- L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- A. Verma et al., "Large-scale cluster management at Google with Borg," *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.
- S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.
- B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

- A. Verma, L. Cherkasova, R. Campbell, and G. Shen, "Large-scale cluster management at google," in *International Conference on Autonomic Computing*, 2008.
- J. Hamilton, "On designing and deploying internet-scale services," *LISA*, 2007.
- L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- A. Verma *et al.*, "Large-scale cluster management at google with borg," in *EuroSys*, 2015.
- S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.
- B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering*. O'Reilly Media, 2016.