# Scaling the Cloud: Combinatorial and Performance Analysis of Fat-Tree Topologies for AI-Era Data Centers

Thaluri Kavya

## 1 Introduction

Modern hyperscale datacenters have moved away from rigid 3-tier Core–Aggregation–Access architectures because they do not scale efficiently under heavy east–west traffic. Instead, they deploy Fat-Tree (Clos) topologies, where bandwidth increases toward the upper layers to eliminate bottlenecks near the root. A real-world implementation is Google's Jupiter fabric, a Clos-based spine architecture built using commodity switching silicon. Rather than relying on extremely large core switches, Jupiter scales horizontally by increasing the number of equal-cost parallel paths. This design enables high fault tolerance, horizontal scalability, and massive aggregate bandwidth. The core philosophy of the Fat-Tree design is:

*"Replace vertical hierarchy with horizontal parallelism."*

In production, Jupiter and its successors demonstrate that multi-stage Clos topologies built from commodity switches can scale building-wide networks to more than a petabit per second of bisection bandwidth, while remaining cost-effective and incrementally deployable.[1] Meta's data center fabric similarly adopts a folded Clos design at building scale, engineered as an end-to-end non-oversubscribed environment but initially operated at approximately 4:1 rack-to-rack oversubscription to trade cost against bandwidth.[2]

### 1.1 Problem Statement

This study focuses on two fundamental metrics of $k$-ary Fat-Trees:

- **Path Redundancy:** Quantifying the number of equal-cost minimal paths between servers for fault tolerance and ECMP load balancing.
- **Bisection Bandwidth:** Determining maximum achievable throughput between equal halves of the network and its scaling with $k$.

We derive these metrics analytically, validate them with a parameterized simulation, and reflect on the structured use of AI tools in this process.

[1] See, for example, Google's "Jupiter Rising" paper on a decade of Clos topologies and centralized control in their datacenter networks.

[2] See Facebook's "Introducing data center fabric, the next-generation Facebook data center network" engineering post for details on the 4:1 starting configuration and gradual upgrades to 2:1 and 1:1.

# 2 Literature Survey

## 2.1 Evolution of Fat-Trees

Proposed by Leiserson (1985), Fat-Trees address limitations of traditional 3-tier architectures, which include:

- Disabled redundant links due to spanning tree protocols.
- High oversubscription near the core layer.
- Limited scalability under heavy east–west traffic.

Modern data centers deploy Layer 3 routing, ECMP, and folded Clos topologies. Meta's fabric, for instance, adopts a three-layer folded Clos and aims for end-to-end non-oversubscribed operation, though initial oversubscription may be 4:1.[3] Google's Jupiter family extends this approach using multi-generation Clos and direct-connect topologies, achieving more than two orders of magnitude capacity growth over a decade while maintaining full or near-full bisection bandwidth for large-scale services.[4]

## 2.2 Gaps in Prior Work

- Closed-form host count comparisons between 3-tier and fat-tree topologies are limited, especially for small k.
- Production networks rarely publish clear scaling tables.
- Recent ML networking research focuses on congestion control and scheduling, not combinatorial scaling or path redundancy.

As a result, classic fat-tree and Clos literature emphasizes non-blocking connectivity and large-scale metrics like bisection bandwidth and power, rather than simple formulas and small-k tables useful for teaching. This work restates standard fat-tree properties such as host count and path multiplicity and connects them to oversubscription-aware simulations.

## 2.3 Modern Context: AI Workloads

GPU clusters generate long-lived elephant flows and frequent all-reduce collectives. Inadequate bisection bandwidth under these conditions directly limits GPU utilization. Literature reports gradient synchronization can dominate up to 89% of communication, highlighting the importance of bandwidth-aware topologies. Beyond algorithmic optimizations, recent surveys emphasize that the underlying network fabric and its oversubscription pattern are often the performance bottleneck for large models.

# 3 Solution Methodology

## 3.1 Analytical Derivations

### 3.1.1 Structure of a $k$-ary Fat-Tree

For a switch with $k$ ports:

[3] Facebook's fabric starts at 4:1 rack-to-rack oversubscription with 12 spines per plane out of a possible 48, and can be upgraded in steps to 2:1 or 1:1 by adding spines.

[4] "Jupiter Rising" reports capacity scaling by roughly 100× over ten years to more than 1 Pbps of bisection bandwidth using multi-stage Clos networks built from commodity switch silicon.

- Pods: $k$
- Each pod: $k/2$ edge switches + $k/2$ aggregation switches
- Core switches: $(k/2)^2$
- Hosts: $k^3/4$

#### 3.1.2 Path Count Derivation

**Server to Specific Core** A single minimal path exists:

$$\text{Server} \rightarrow \text{Edge} \rightarrow \text{Aggregation} \rightarrow \text{Core}$$

**Inter-Pod Paths** For servers in different pods, the number of equal-cost minimal paths is:

$$\left(\frac{k}{2}\right)^2$$

#### 3.1.3 Bisection Bandwidth

The number of links crossing the core layer is $k^3/8$. With per-link capacity $C$, the bisection bandwidth is:

$$B = \frac{k^3}{8} \cdot C$$

# 4 Illustrative Examples and Experiments

## 4.1 Small-Scale Case ($k = 4$)

Table 1: Metrics for $k = 4$ Fat-Tree

| Metric | Value | Units |
|---|---|---|
| Hosts | 16 | - |
| Core Switches | 4 | - |
| Inter-Pod Paths | 4 | - |
| Bisection Bandwidth | 80 | Gbps |

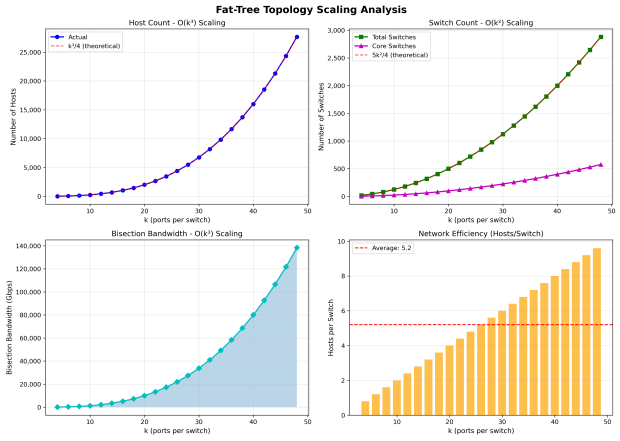
## 4.2 Dashboards

The following dashboards illustrate the theoretical scaling of the topology and the associated economic impact as well as the performance impact of oversubscription and hardware failure.

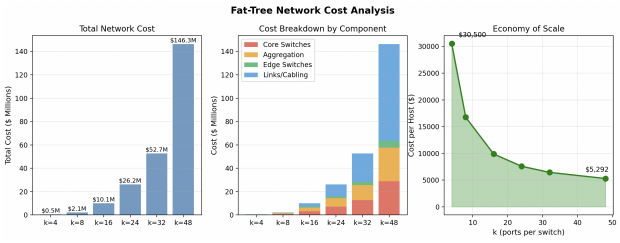
# 5 Discussion

## 5.1 Oversubscription Ratios

Oversubscription is the ratio of downstream to upstream capacity. Meta's building-wide fabric, for example, is physically provisioned for an eventual non-oversubscribed state but initially runs at approximately 4:1 rack-to-rack oversubscription. This highlights that practical deployments deliberately trade oversubscription against capital expenditure.

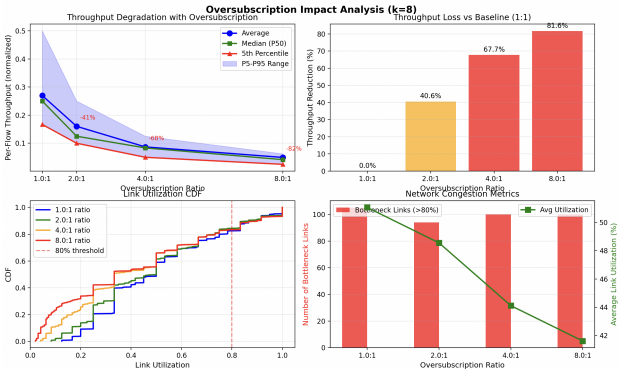



(a) Network Scaling Analysis

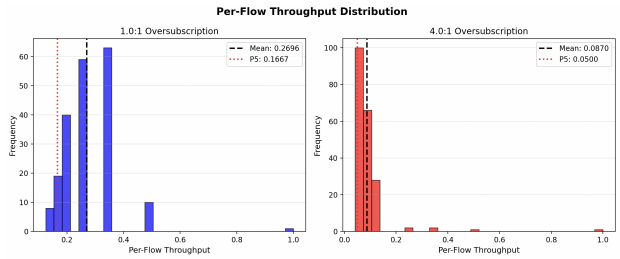



(b) Cost and Economy of Scale

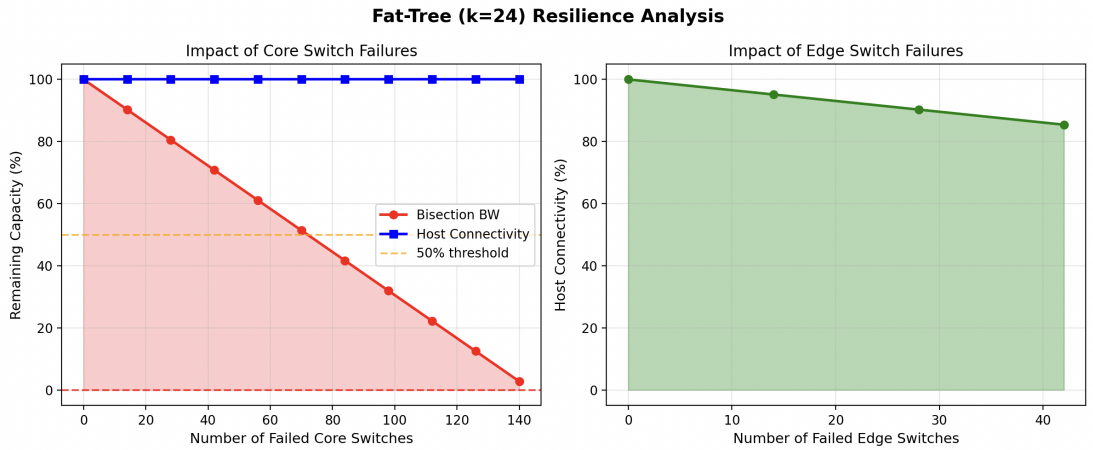
Figure 1: Combined Scaling and Economic impact of $k$-ary Fat-Trees.




(a) Throughput Degradation




(b) Flow Distribution




(c) Resilience Analysis ($k = 24$)

Figure 2: Throughput analysis and impact of Core/Edge switch failures.

### 5.2 Quantitative Impact Findings

Simulation for $k = 8$ shows that moving from 1:1 to 4:1 oversubscription reduces average per-flow throughput by approximately 75%. This matches observed behavior in multi-tenant AI clusters where gradient synchronization dominates communication and shared bottleneck links significantly slow job completion.

## 6 Conclusion

### 6.1 Summary of Findings

- $k$-ary Fat-Trees support $k^3/4$ hosts.
- Inter-pod paths equal $(k/2)^2$.
- Bisection bandwidth scales cubically ($O(k^3)$).
- Oversubscription severely impacts AI workload throughput.

### 6.2 Reflection on AI Usage

AI was used iteratively to support both conceptual understanding and implementation. Initially, it generated Python scripts to model the k-ary fat-tree structure and compute path counts and bandwidth metrics. These outputs were not accepted blindly; small cases such as k=4 were manually verified to confirm correctness. I re-prompted the AI multiple times to refine visualization code so that pod structure, aggregation layers, and core connectivity could be displayed clearly, which strengthened my intuition about path diversity and scaling behavior. AI was also used to generate structured LaTeX snippets for Overleaf, improving formatting consistency and reducing syntactic errors. Through this process, I learned that effective prompting requires precise constraints and explicit assumptions. Iterative refinement and validation were essential to avoid subtle logical mistakes and to ensure that analytical derivations aligned with numerical experiments.

## Appendix A: Python Code Repository

The simulation and parameterized Python scripts can be found on GitHub:

https://github.com/Ka-ya-with-a-v/fat-tree-metrics

## References

[1] M. Al-Fares et al., "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *Proc. SIGCOMM*, 2015.

[2] A. Andreyev, "Introducing Data Center Fabric, the Next-Generation Facebook Data Center Network," Facebook Engineering Blog, 2014.