# Analysis of Data Center Network Topologies: Path Diversity, Bisection Bandwidth, and the Fat Tree Visualizer

Au Ze Hong, Maxwell
Cloud Computing — Assignment 1 (Question 5)

February 2026

## 1 Introduction and Literature Survey

Modern data centers rely on structured network topologies to deliver high bisection bandwidth and multipath redundancy to thousands of servers. The dominant design paradigm is the *Clos network*, introduced by Charles Clos in 1953 [2], which arranges switches in multiple stages so that any input can reach any output through several independent paths.

Al-Fares et al. [3] demonstrated that a $k$*-ary fat tree*—a folded Clos built entirely from commodity $k$-port switches—can deliver full bisection bandwidth at a fraction of the cost of proprietary chassis switches. This work became the reference architecture for academic study of data center networks.

Three hyperscale operators have published detailed accounts of their production networks:

- **Google Jupiter** [4, 5]: a multi-stage Clos interconnected through Optical Circuit Switches (OCS), evolving from 10 Gbps to 400 Gbps per port across a decade.
- **Amazon (AWS)** [6]: a leaf-spine (2-tier Clos) fabric where every leaf switch connects to every spine switch, as described in AWS re:Invent 2022.
- **Meta (Facebook)** [7, 8]: a 3-level Clos with spine, fabric, and ToR layers organised into pods, running standard BGP4.

The open-source *Fat Tree Visualizer* by H. Liu [1] provides an interactive tool for exploring fat tree topologies. In this assignment we fork the visualizer and extend it with three additional topology modules (Jupiter, Amazon, Meta), a BFS-based path analysis engine, and an exact bisection bandwidth algorithm, then use the tool to answer each sub-question of Question 5.

## 2 Solution Methodology

### 2.1 Topology Construction

Each topology is constructed as an undirected graph $G = (V, E)$ where $V$ is partitioned into *hosts* $H$ and *switches* $S$. Table 1 summarises the four architectures.

Table 1: Topology parameters for the four data center architectures.

| | **Fat Tree** | **Jupiter** | **Amazon** | **Meta** |
|---|---|---|---|---|
| Hierarchy | $d$-level Clos | 3-level + OCS | 2-tier Clos | 3-level Clos |
| Parameters | $k$, $d$ | 4 blocks, 4 OCS | 4 spine, 8 leaf | 4 spine, 4 pods |
| Hosts | $k^{d-1} \cdot k$ | 32 | 16 | 32 |
| Switches | $(2d-1)\, k^{d-1}$ | 24 | 12 | 28 |
| Links | $d \cdot k \cdot k^{d-1}$ | 96 | 48 | 96 |
| Root type | Level-0 switches | OCS | Spine | Spine |

Note: for the fat tree, hosts and links above count one side of the symmetric visual layout, which is the graph used for analysis. The statistics panel displays the full (both-sides) count $2k^{d-1} \cdot k$.

### 2.2 Path Counting via BFS

To answer parts (a) and (b), we count shortest paths using a standard BFS-based algorithm. Starting from a source node $s$, we maintain for every visited node $v$: its distance $\mathrm{dist}(v)$ and the number of shortest paths $\mathrm{count}(v)$. When a neighbour $u$ of $v$ is first discovered, $\mathrm{count}(u) \leftarrow \mathrm{count}(v)$; when $u$ is reached again at the same distance, $\mathrm{count}(u) \mathrel{+}= \mathrm{count}(v)$. This runs in $O(|V| + |E|)$ per query.

For paths through a specific switch $w$, we use the *product principle*: if $w$ lies on a shortest $s$–$t$ path (i.e. $\mathrm{dist}(s,w) + \mathrm{dist}(w,t) = \mathrm{dist}(s,t)$), then the number of shortest $s$–$t$ paths through $w$ is $\mathrm{count}(s \to w) \times \mathrm{count}(w \to t)$.

## 2.3 Bisection Bandwidth

**Fat tree (exact formula).** For a $k$-ary fat tree of depth $d$, the minimum bisection cut is at the root level: each of the $k^{d-1}$ root switches has $k$ links to each half, giving:

$$B_{\text{fat tree}} = k^{d-1} \times k = k^d \text{ link-units.}$$

**General topologies (exact enumeration).** For the other three topologies we compute the exact bisection bandwidth by:

1. Grouping hosts by their parent ToR/leaf switch.
2. Enumerating all subsets of groups whose total host count equals $\lfloor |H|/2 \rfloor$ (via bitmask; feasible for $\leq 20$ groups).
3. For each balanced host partition, greedily assigning each switch to the side with more of its neighbours (iterating to convergence).
4. Returning the minimum number of crossing links across all partitions.

This replaces the original visualizer's spatial heuristic (sorting by $x$-coordinate) with a provably tighter bound.

## 3 Illustrative Examples and Results

### 3.1 Part (a): Server to Root Switch Paths

Table 2 shows the number of minimal-length shortest paths from an arbitrary server (M1) to representative root switches across all four topologies.

Table 2: Shortest paths from server M1 to root switches.

| **Topology** | **Pair** | **Paths** | **Length** | **Derivation** |
|---|---|---|---|---|
| Fat Tree ($k$=4, $d$=3) | M1 → S1 | 16 | 3 | $k^{d-1} = 4^2 = 16$ |
| Google Jupiter | M1 → S1 (OCS) | 2 | 3 | BFS (see below) |
| Amazon Leaf-Spine | M1 → S1 (Spine) | 1 | 2 | BFS (see below) |
| Meta 3-Level Clos | M1 → S1 (Spine) | 2 | 3 | BFS (see below) |

**Why fat tree admits a closed-form formula.** The $k$-ary fat tree has a *uniform recursive structure*: at every intermediate switch level between a server and the root, the topology fans out to exactly $k$ parent switches, each of which is equally valid for reaching any root switch. A server at depth $d-1$ must traverse $d-1$ switch-to-switch hops to reach a level-0 root, and at each hop there are $k$ upward choices that all remain on a shortest path. This gives $k^{d-1}$ minimal-length paths by a simple product argument.

**Why the other topologies lack a closed-form.** Jupiter, Amazon, and Meta do not have this uniform fan-out structure. Their path counts depend on the *specific* (server, root) pair chosen:

- **Amazon Leaf-Spine:** Each host connects to exactly one leaf switch, which connects to all 4 spines. The path M1 → Leaf → Spine is unique for a given spine, so there is always exactly **1 path** (length 2). There is no branching because the host has a single parent.
- **Jupiter & Meta:** The path Host → ToR → {Agg/Fabric} → Root has a branching factor equal to the number of aggregation/fabric switches per block/pod. With 2 per block/pod, every server has exactly **2 paths** (length 3) to any given root.

These counts can be verified by hand for a fixed configuration but do not generalise to a single formula when the topology parameters change independently. We therefore compute them algorithmically using the BFS shortest-path counting algorithm described in Section 2.2.

**Key insight:** The fat tree provides significantly more path diversity ($k^{d-1}$ paths) to any single root switch than the production topologies, which have 1–2 paths. This is because the fat tree has a wider fan-out at each level, while production topologies use fewer, higher-radix switches.

### 3.2 Part (b): Host-to-Host Paths

Table 3 presents the path counts for the requested host pairs (M1–M3, M1–M5) and the constrained paths through switch S2. For Jupiter and Meta, we also include an *inter-group* pair (M1–M9) where the hosts are in different blocks/pods.

**Key insight:** For Jupiter and Meta, same-block/pod hosts never traverse the top-level switches (OCS or spine) on shortest paths, yielding 0 paths through S2. This demonstrates *traffic locality*—intra-pod traffic stays within the pod. Only inter-block/pod pairs (e.g., M1–M9) exercise the full path diversity through the top tier. Amazon's flat leaf-spine has no such distinction: every cross-leaf pair traverses all 4 spines equally.

Table 3: Shortest paths between host pairs across all topologies.

| Topology | Pair | Paths | Len. | Notes |
|---|---|---|---|---|
| Fat Tree ($k$=4) | M1 → M3 | 4 | 4 | Same edge switch |
| | M1 → M5 | 16 | 6 | Different edge switches |
| | M1 → M5 via S2 | 1 | 6 | S2 is a root switch |
| Jupiter | M1 → M3 | 2 | 4 | Same block |
| | M1 → M5 | 2 | 4 | Same block |
| | M1 → M5 via S2 | 0 | — | S2 (OCS) not on intra-block path |
| | M1 → M9 | 8 | 6 | *Inter-block, via OCS* |
| Amazon | M1 → M3 | 4 | 4 | Different leaves |
| | M1 → M5 | 4 | 4 | Different leaves |
| | M1 → M5 via S2 | 1 | 4 | S2 is a spine switch |
| Meta | M1 → M3 | 2 | 4 | Same pod |
| | M1 → M5 | 2 | 4 | Same pod |
| | M1 → M5 via S2 | 0 | — | S2 (spine) not on intra-pod path |
| | M1 → M9 | 8 | 6 | *Inter-pod, via spine* |

## 3.3 Part (c): Bisection Bandwidth

Table 4: Bisection bandwidth (in link-units) for all topologies.

| Topology | BW | Method | Explanation |
|---|---|---|---|
| Fat Tree ($k$=4, $d$=3) | 64 | Formula $k^d$ | 16 root switches × 4 links each |
| Google Jupiter | 16 | Exact enum. | Cut between blocks 2 and 3 |
| Amazon Leaf-Spine | 16 | Exact enum. | 4 leaves × 4 spines crossing |
| Meta 3-Level Clos | 16 | Exact enum. | Cut between pods 2 and 3 |

**Key insight:** The fat tree with $k$=4 achieves 4× the bisection bandwidth of the production topologies at this scale. The three production topologies all converge to the same bisection bandwidth of 16 link-units, despite their different architectures—this reflects the common design principle of matching cross-sectional capacity at the top tier.

## 3.4 Part (d): Scaling Experiments

We vary the fat tree parameter $k$ to observe how network metrics scale:

Table 5: Fat tree scaling ($d$=3) across different $k$ values.

| $k$ | Hosts | Switches | Links | Bisection BW | Paths (M1→root) |
|---|---|---|---|---|---|
| 2 | 8 | 10 | 24 | 8 | 4 |
| 4 | 64 | 80 | 192 | 64 | 16 |
| 8 | 512 | 640 | 1,536 | 512 | 64 |
| 16 | 4,096 | 5,120 | 12,288 | 4,096 | 256 |

All metrics scale as powers of $k$: hosts as $k^3$, switches as $5k^2$, bisection bandwidth as $k^3$, and path diversity as $k^2$. This confirms the well-known $O(k^3)$ scaling of fat trees, which is precisely the property that makes them attractive for data centers where the cost of commodity switches grows linearly with $k$.

# 4 Use of AI and Vibe Coding

This project was developed using **Claude Code** (Anthropic's Claude Opus 4.6) as an AI copilot throughout the process. The interaction followed a structured workflow:

1. **Correctness audit.** The AI was asked to "assess the correctness of the analysis for all supported architectures." It read every source file, manually traced the topology construction for each architecture, verified the BFS algorithm, and identified three concrete bugs:
   - The fat tree's `drawHosts()` drew a hardcoded 3 hosts per edge switch regardless of $k$, causing the graph to have fewer hosts than the formula predicted.
   - The bisection bandwidth for non-fat-tree topologies used a spatial heuristic (sort by $x$-coordinate, split in half) that could miss the true minimum cut.
   - The analysis only showed intra-pod host pairs for Jupiter and Meta, always producing 0 paths through top-level switches—correct but uninformative.
2. **Bug fixing.** Each issue was fixed with targeted edits: the host loop now iterates $k$ times; bisection uses exact balanced-partition enumeration; and an inter-group host pair finder was added.

3. **Prompt engineering lessons.** Vague prompts like "check if it works" produced generic responses. The effective prompt was specific: "assess the correctness of the analysis for the assignment . . . for all supported architectures." This directed the AI to read every topology file and trace computations end-to-end rather than giving surface-level feedback.
4. **Source attribution.** The AI was prompted to find the original papers/talks behind each topology, which were added both as code comments and as frontend references.

The key lesson was that AI copilots are most effective when given a precise audit scope and when the user validates the AI's reasoning against the actual code rather than trusting outputs uncritically.

## 5 Discussion

**Path diversity vs. practical routing.** While the fat tree offers $k^{d-1}$ shortest paths to any root switch, production networks (Jupiter, Amazon, Meta) provide only 1–2 paths per root. This does not necessarily make them inferior: production networks achieve load balancing through ECMP across *all* root switches simultaneously, whereas our analysis counts paths to a *single* root. The total multipath capacity across all roots is comparable.

**Locality as a design feature.** Jupiter and Meta's 0-path result through top-level switches for intra-pod pairs is a deliberate design feature, not a limitation. By keeping intra-pod traffic local, these architectures reduce load on expensive inter-pod links and lower latency for rack-local communication—critical for distributed training workloads where GPUs within a rack communicate intensively.

**Bisection bandwidth convergence.** All three production topologies yield identical bisection bandwidth (16 link-units) despite different structures. This suggests that at equivalent scale, the choice between 2-tier (Amazon) and 3-tier (Meta, Jupiter) architectures is driven more by operational considerations (failure domains, incremental expansion) than raw bandwidth capacity.

**Limitations.** Our analysis treats all links as having equal bandwidth (1 link-unit). In practice, inter-pod links may run at higher speeds (e.g., 400G vs. 100G), and OCS in Jupiter can be reconfigured to shift bandwidth dynamically. The exact bisection algorithm, while correct for small topologies, is exponential in the number of host groups and would need approximation for larger networks.

## 6 Conclusion

We extended the Fat Tree Visualizer with three production-inspired datacenter topologies (Google Jupiter, Amazon Leaf-Spine, Meta 3-Level Clos) and a BFS-based analysis engine that answers all parts of Question 5 across all four architectures. Key findings:

- Fat trees provide $k^{d-1}$ shortest paths per server-root pair and bisection bandwidth of $k^d$, both scaling polynomially.
- Production topologies trade path diversity for operational simplicity, with traffic locality keeping intra-pod communication off the spine.
- All three production topologies converge to the same bisection bandwidth at equivalent scale, suggesting that topology choice is driven by operational rather than capacity concerns.
- AI copilots proved effective for code auditing and bug detection when given precise, scoped prompts, but required human verification of their reasoning.

The full source code is available as an appendix. The original visualizer is at https://github.com/h8liu/ftree-vis; our extended fork with the analysis engine and additional topologies accompanies this submission.

## References

[1] H. Liu, "Fat Tree Visualizer," GitHub repository. https://github.com/h8liu/ftree-vis

[2] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, 2008, pp. 63–74.

[4] A. Singh et al., "Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network," in *Proc. ACM SIGCOMM*, 2015, pp. 183–197. https://dl.acm.org/doi/10.1145/2785956.2787508

[5] S. Poutievski et al., "Jupiter evolving: Transforming Google's datacenter network via optical circuit switches and software-defined networking," in *Proc. ACM SIGCOMM*, 2022. https://research.google/pubs/jupiter-evolving-transforming-googles-datacenter-network-via-optical-circuit-switches-and-software-

[6] JR Rivers and S. Callaghan, “Dive deep on AWS networking infrastructure,” AWS re:Invent 2022 (NET402). https://d1.awsstatic.com/events/Summits/reinvent2022/NET402_Dive-deep-on-AWS-networking-infrastructure.pdf

[7] A. Andreyev, “Introducing data center fabric, the next-generation Facebook data center network,” Meta Engineering Blog, Nov. 2014. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/

[8] A. Andreyev et al., “Reinventing Facebook’s data center network with F16 and Minipack,” Meta Engineering Blog, Mar. 2019. https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/

## A Key Source Code

### A.1 BFS Shortest Path Counting (analysis.js)

```
function countShortestPaths(adj, src, dst) {
  var dist = {}, count = {};
  var queue = [src];
  dist[src] = 0;  count[src] = 1;
  while (queue.length > 0) {
    var node = queue.shift();
    var neighbors = adj[node] || [];
    for (var i = 0; i < neighbors.length; i++) {
      var nb = neighbors[i];
      if (dist[nb] === undefined) {
        dist[nb] = dist[node] + 1;
        count[nb] = count[node];
        queue.push(nb);
      } else if (dist[nb] === dist[node] + 1) {
        count[nb] += count[node];
      }
    }
  }
  return { count: count[dst] || 0, dist: dist[dst] || -1 };
}
```

### A.2 Paths Through a Specific Node (analysis.js)

```
function countPathsThroughNode(adj, src, dst, via) {
  var srcToDst = countShortestPaths(adj, src, dst);
  var srcToVia = countShortestPaths(adj, src, via);
  var viaToDst = countShortestPaths(adj, via, dst);
  if (srcToVia.dist + viaToDst.dist === srcToDst.dist) {
    return srcToVia.count * viaToDst.count;
  }
  return 0;
}
```

### A.3 Exact Bisection Bandwidth (analysis.js)

```
// Enumerate balanced host-group partitions, assign switches greedily
for (var mask = 1; mask < (1 << nGroups) - 1; mask++) {
  var leftCount = 0;
  for (var g = 0; g < nGroups; g++)
    if (mask & (1 << g)) leftCount += groupSizes[g];
  if (leftCount !== halfHosts) continue;
  // ... assign switches to minimize crossing links ...
  // ... track minimum cut across all partitions ...
}
```

### A.4 Fat Tree Host Generation Fix (ftree.js)

```
function drawHosts(list, y, direction) {
  for (var i = 0; i < list.length; i++) {
    var switchId = switchMap[depth - 1 + "_" + i].id;
    // Draw exactly k hosts per edge switch (was hardcoded to 3)
    var spacing = k <= 1 ? 0 : 8 / (k - 1);
    var totalSpan = k <= 1 ? 0 : 8;
    for (var h = 0; h < k; h++) {
      var dx = k <= 1 ? 0 : -totalSpan / 2 + h * spacing;
      drawHost(list[i], y, hhost * direction, dx, switchId);
    }
  }
}
```