Understanding Lifecycle Management Complexity of Datacenter Topologies

Mingyang Zhang, ***University of Southern California;***
Radhika Niranjan Mysore, ***VMware Research;*** **Sucha Supittayapornpong and Ramesh Govindan,** ***University of Southern California***

https://www.usenix.org/conference/nsdi19/presentation/zhang

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

NetApp®

Understanding Lifecycle Management Complexity of Datacenter Topologies

Mingyang Zhang *USC*
Radhika Niranjan Mysore *VMWare*
Sucha Supittayapornpong *USC*
Ramesh Govindan *USC*

Abstract

Most recent datacenter topology designs have focused on performance properties such as latency and throughput. In this paper, we explore a new dimension, life cycle management complexity, which attempts to understand the complexity of deploying a topology and expanding it. By analyzing current practice in lifecycle management, we devise complexity metrics for lifecycle management, and show that existing topology classes have low lifecycle management complexity by some measures, but not by others. Motivated by this, we design a new class of topologies, FatClique, that, while being performance-equivalent to existing topologies, is comparable to, or better than them by *all* our lifecycle management complexity metrics.

1 Introduction

Over the past decade, there has been a long line of work on designing datacenter topologies [2, 35, 31, 32, 3, 4, 20, 1]. While most have focused on performance properties such as latency and throughput, and on resilience to link and switch failures, datacenter *lifecycle management* [30, 38] has largely been overlooked. Lifecycle management is the process of building a network, physically deploying it on a data-center floor, and expanding it over several years so that it is available for use by a constantly increasing set of services.

With datacenters living on for years, sometimes up to a decade [31, 12], their lifecycle costs can be high. A data center design that is hard to deploy can stall the rollout of services for months; this can be expensive considering the rate at which network demands have historically increased [31, 23]. A design that is hard to expand can leave the network functioning with degraded capacity impacting the large array of services that depend on it.

It is therefore desirable to commit to a data-center network design only after getting a sense of its lifecycle management cost and complexity over time. Unfortunately, the costs of the large array of components needed for deployment such as switches, transceivers, cables, racks, patch panels[1], and cable trays, are proprietary and change over time, and so are hard to quantify. An alternative approach is to develop *complexity measures* (as opposed to dollar costs) for lifecycle management, but as far as we know, no prior work has addressed this. In part, this is due to the fact that intuitions about lifecycle management are developed over time and with operations experience, and these lessons are not made available universally. Unfortunately, in our experience, this lack of a clear understanding of lifecycle management complexity often results in costly mistakes in the design of datacenters that are discovered during deployment and therefore cannot be rectified. Our paper is a first step towards useful characterizations of lifecycle management complexity.

Contributions. To this end, our paper makes three contributions. First, we design several *complexity metrics* (§3 and §4) that can be indicative of lifecycle management costs (*i.e.,* capital expenditure, time and manpower required). These metrics include the number of: switches, patch panels, *bundle-types*, expansion steps, and links to be re-wired at a patch panel rack during an expansion step.

We design these metrics by identifying structural elements of network deployments that make their deployment and expansion challenging. For instance, the number of switches in the topology determines how complex the network is in terms of *packaging* – laying out switches into homogeneous racks in a space efficient manner. Wiring complexity can be assessed by the number of cable *bundles* and the patch panels a design requires. As these increase, the complexity of manufacturing and packaging all the different cable bundles efficiently into cable trays, and then routing them from one patch panel to the next can be expected to increase. Finally, because expansion is carried out in steps [38], where the network operates at degraded capacity at each step, the number of expansion steps is a measure of the reduced availability in the network induced by lifecycle management. Wiring patterns also determine the number of links that need to be rewired at a patch panel during each step of expansion, a measure of step complexity [38].

Our second contribution is to use these metrics to compare the lifecycle management costs of two main classes of datacenter topologies recently explored in the research literature (§2), Clos [2] and expander graphs [32, 35]. We find that neither class dominates the other: Clos has relatively lower wiring complexity; its symmetric design leads to more uniform bundling (and fewer cable bundle types); but expander graphs at certain scales can have simpler packaging requirements due to their edge expansion property [32]; they end up using much fewer switches than Clos to achieve the same network capacity. Expander graphs also demonstrate better expansion properties because they have *fat edges* (§4) which permit more links to be rewired in each step.

Finally we design and synthesize a novel and practical class of topologies called FatClique (§5), that has lower overall lifecycle management complexity compared to Clos and expander graphs. We do this by combining favorable design

[1] A patch panel or a wiring aggregator is a device that simplifies cable re-wiring.

elements from these two topology classes. By design, FatClique incorporates 3 levels of hierarchy and uses a clique as a building block while ensuring edge expansion. At every level of its hierarchy, FatClique is designed to have fat edges, for easier expansion, while utilizing much fewer patch panels and therefore inter-rack cabling.

Evaluations of these topology classes at three different scales, the largest of which is 16× the size of Jupiter, shows that FatClique is the best at most scales by *all* our complexity metrics. It uses 50% fewer switches and 33% fewer patch panels than Clos at large scale, and has a 23% lower cabling cost (an estimate we are able to derive from published cable prices). Finally, FatClique can permit fast expansion while degrading network capacity by small amounts (2.5-10%): at these levels, Clos can take 5 × longer to expand the topology.

2 Background

Data center topology families. Data centers are often designed for high throughput, low latency and resilience. Existing data center designs can be broadly classified into the following families: (a) Clos-like tree topologies, *e.g.,* Google's Jupiter [31], Facebook's fbfabric [3], Microsoft's VL2 [13], F10 [22]; (b) Expander graph based topologies, *e.g.,* Jellyfish [32], Xpander [35]; (c) 'Direct' topologies built from multi-port servers, *e.g.,* BCube [14], DCell [15]. (d) Low diameter, strongly-connected topologies that rely on high-radix switches, *e.g.,* Slimfly [4], Dragonfly [20]; (e) Reconfigurable optical topologies like Rotornet and ProjecToR [24, 9, 11, 16, 39].

Of these, Clos and Expander based topologies have been shown to scale using widely deployed merchant silicon. The ecosystem around the hardware used by these two classes, *e.g.,* cabling, cable trays used, rack sizes, is mature and well-understood, allowing us to quantify some of the operational complexity of these topologies.

Direct multi-port server topologies and some reconfigurable optical topologies [24, 11, 16, 39] rely on newer hardware technologies that are not mainstream yet. It is hard to quantify the operational costs of these classes without making significant assumptions about such hardware. Low diameter topologies like Slimfly [4] and Dragonfly [20], can be built with hardware that is available today, but they require strongly connected groups of switches. Their incremental expansion comes at high cost and complexity; high-radix switches either need to be deployed well in advance, or every switch in the topology needs to be upgraded during expansion, to preserve low diameter.

To avoid estimating operational complexity of topologies that rely on new hardware, or on topologies that unacceptably constrain expansion, we focus on the Clos and Expander families.

Clos. A logical Clos topology with N servers can be constructed using switches with radix k connected in $n = \log_{\frac{k}{2}}(\frac{N}{2})$ layers based on a canonical recursive algorithm in [36][2]. Fattree [2] and Jupiter [31] are special cases of Clos topology with 3 and 5 layers respectively. Clos construction naturally allows switches to be packaged together to form a chassis [31]. Since there are no known generic Clos packaging algorithm that can help design such a chassis, for a Clos of any scale, we designed one to help our study of its operational complexity. We present this algorithm in §A.1.

Expander graphs. Jellyfish and Xpander benefit from the high *edge expansion* property of expander graph to use a near optimal number of switches, while achieving the same bisection bandwidth as Clos based topologies [35]. Xpander splits N servers among switches by attaching s servers to each switch. With a k port switch, the remaining ports $p = k - s$ are connected to other switches that are organized in p blocks called *metanodes*. Metanodes are a group of switches, containing $l = N/(s \cdot (p+1))$ switches, which increase as topology scale N increases. There are no connections between the switches of a metanode. Jellyfish is a degree bounded random graph (see [32] for more details).

Takeaway. A topology with high edge expansion [35] can achieve a target capacity with fewer switches, leading to lower overall cost.

3 Deployment Complexity

Deployment is the process of realizing a physical topology in a data center space (*e.g.,* a building), from a given logical topology. Deployment complexity can be reduced by careful *packaging, placement* and *bundling* strategies [31, 20, 1].

3.1 Packaging, Placement, and Bundling

Packaging of a topology involves careful arrangement of switches into racks, while placement involves arranging these racks into rows on the data center floor. The spatial arrangement of the topology determines the type of cables needed between switches. For instance, if two connected switches are within the same rack, they can use short-range cheaper copper cables, while connections between racks require more expensive optical cables. Optical cable costs are determined by two factors: the cost of transceivers and the length of cables (§3.2). Placement of switches on the datacenter floor can also determine costs: connecting two switches placed at two ends of the data center building might require long range cables and high-end transceivers.

Chassis, racks, and blocks. Packaging connected switches into a single chassis using a backplane completely removes the need for physical connecting cables. At scale, the cost and complexity savings from using a chassis-backplane can be significant. One or more chassis that are interconnected can be packed into racks such that: (a) racks are as homogeneous as possible, *i.e.,* a topology makes use of only a few types of racks to simplify manufacturing and (b) racks are packed as

[2] This equation for n can be used to build a Clos with 1:1 oversubscription. For a Clos with an over-subscription x:y we would need $n = \log_{\frac{k}{2}}(\frac{y \cdot N/x}{2})$ layers.

densely as possible to reduce space wastage. Some topologies define larger units of co-placement and packaging called *blocks*, which consist of groups of racks. Examples of blocks include pods in Fattree. External cabling from racks within a block are routed to wiring aggregators (*i.e.,* patch panels [25]) to be routed to other blocks. For blocks to result in lower deployment complexity, three properties must be met: (a) the ports on the patch panel that it connects to are not wasted, when the topology is built out to full scale, (b) wiring out of the block should be as uniform as possible, and (c) racks in a block must be placed close to each other to reduce the length and complexity of wiring.

Bundling and cable trays. When multiple fibers from the same set of physically adjoint (or neighboring) racks are destined to another set of neighboring racks, these fibers can be *bundled* together. A fiber *bundle* is a fixed number of identical-length fibers between two clusters of switches or racks. Manufacturing bundles is simpler than manufacturing individual fibers, and handling such bundles significantly simplifies operation complexity. Cable bundling reduces capex and opex by around 40% in Jupiter [31].

Patch panels facilitate bundling since the patch panel represents a convenient aggregation point to create and route bundles from the set of fibers destined to the same patch panel (or the same set of physically proximate patch panels). Figure 1 shows a Clos topology instance (left) and its physical realization using patch panels (right). Each aggregation block in the Clos network connects with one link to each spine block. The figure on the right shows how these links are routed physically. Bundles with two fibers each from two aggregations are routed to two (lower) patch panels. At each patch panel, these fibers are rebundled, by grouping fibers that go to the same spine in new bundles, and routed to two other (upper) patch panels that connect to spines. The bundles from the upper patch panels are then routed to the spines. Figure 1 assumes that patch panels are used as follows: bundles are connected to both the front and back ports on patch panels. For example, bundles from the aggregation layer connect to front ports on patch panels and bundles from spines connect to the back ports of patch panels. This enables bundle aggregation and rebundling and simplifies topology expansion.[3]

Bundles and fibers are routed through the datacenter on *cable trays*. The cables that aggregate at a patch panel rack must be routed overhead by using over-row and cross-row trays [26]. Trays have capacity constraints [34], which can constrain rack placement, block sizes, and patch panel placement. Today, trays can support at most a few thousand fibers [34].

[3] [38]'s usage of patch panels is slightly different. All bundles are connected to front ports of patch panels and links are established using jumper cables between the back ports of patch panels. For patch panels of a given port count, both approaches require the same number of patch panels. Our approach enables bundling closer to the aggregation and spine layers; [38] does not describe how bundling is accomplished in their design.

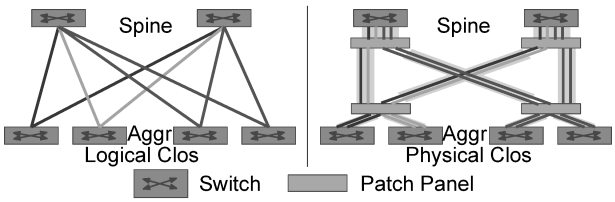


Figure 1: Fiber Re-bundling for Clos at Patch Panels

With current rack and cable tray sizes, a single rack of patch panels can be accommodated by four overhead cable trays, arranged in four directions. In order to avoid aggregating too many links into a single location, it is desirable to space such patch panels apart to accommodate more cable trays. This consideration in turn constrains block sizes; if cables from blocks must be all routed locally, it is desirable that a block only connect to a single rack of patch panel.

3.2 Deployment Complexity Metrics

Based on the previous discussion, we identify several metrics that quantify the complexity of the two aspects of datacenter topology deployment: packaging and placement. In the next subsection, we use these metrics to identify differences between Clos and Expander graph topology classes.

Number of Switches. The total number of switches in the topology determines the capital expenditure for the topology, but it also determines the packaging complexity (switches need to be packed to chassis and racks) and the placement complexity (racks need to be placed on the datacenter floor).

Number of Patch panels. By acting as bundle waypoints, the number of patch panels captures one measure of wiring complexity. The more the number of patch panels, the shorter the cable lengths from switches to the nearest patch panel, but the fewer the bundling opportunities, and vice versa. The number of patch panels needed is a function of topological structure. For instance, in a Clos topology, if an aggregation layer fits into one rack or a neighboring set of racks, a patch panel is not needed between the ToR and the aggregation layer. However, for larger Clos topologies where an aggregation block can span multiple racks, ToR to aggregation links may need to be rebundled through a patch panel. We discuss this in detail in §6.2.

Number of Bundle Types. The number of patch panels alone does not capture wiring complexity. The other measure is the number of distinct *bundle types*. A bundle type is represented by a tuple of (a) the *capacity* of the number of fibers in the bundle, and (b) the *length* of the bundle. If a topology requires only a small number of bundle types, its bundling is more homogeneous; manufacturing and procuring such bundles is significantly simpler, and deploying the topology is also simplified since fewer bundling errors are likely with fewer types.

These complexity measures are *complete*. The number of cable trays, the design of the chassis, and the number of racks can be derived from the number of switches (and the number of servers and the datacenter floor dimensions, which

Topology	4-layer Clos (Medium)	Jellyfish
#servers	131,072	131,072
#switches	28,672	16,384
#bundle types	74	1577
#patch panels	5546	7988

Table 1: Deployment Complexity Comparison

are inputs to the topology design). The number of cables and transceivers can be derived from the number of patch panels.

In some cases, a metric is related to another metric, but not completely subsumed by it. For example, the number of switches determines rack packaging, which only partially determines the number of transceivers per switch. The other determinant of this quantity is the *connectivity in the logical topology* (which switch is connected to which other switch). Similarly, the number of patch panels can influence the number of bundle types, but these are also determined by logical connectivity.

3.3 Comparing Topology Classes

To understand how the two main classes of topologies compare by these metrics, we apply these to a Clos topology and to a Jellyfish topology that support the same number of servers (131,072) and the same bisection bandwidth. This topology corresponds to twice the size of Jupiter. In §6, we perform a more thorough comparison at larger and smaller scales, and we describe the methodology by which these numbers were generated.

Table 1 shows that the two topology classes are qualitatively different by these metrics. Consistent with the finding in [32], Jellyfish only needs a little over half the switches compared to Clos to achieve comparable capacity due to its high edge expansion property. But, by other measures, Clos performs better. It exposes far fewer ports outside the rack (a little over half that of Jellyfish); we say Clos has better *port-hiding*. A pod in this Clos contains 16 aggregation and 16 edge switches[4]. The aggregation switches can be can be packed into a single rack, so bundles from edge switches to aggregation switches do not need to be rebundled though patch panels, and we only need two layers of patch panels between aggregation and spine layer. However, in Jellyfish, almost all links are inter-rack links, so it requires more patch panels.

Moreover, for Clos, since each pod has the same number of links to each spine, all bundles in Clos have the same capacity (number of fibers). However, the length of bundles can be different, depending on the relative placement of the patch panels between aggregation and spine layers, so Clos has 74 bundle types. However, since Jellyfish is a purely random graph without structure, to enable bundling, we group a fixed amount of neighbor racks as blocks to enable bundling. Since connectivity is random, the number of links between blocks are not uniform, Jellyfish needs almost 20× the number of bundle types. In §6, we show that Xpander also has qualitatively similar behavior in large scale.

Takeaway. Relative to a structured hierarchical class of topologies like Clos, the expander graph topology has inherently higher deployment complexity in terms of the number of bundle types and cannot support port-hiding well.

[4] we follow the definition of pod in [2].

4 Topology Expansion

The second important component of topology lifecycle management is *expansion*. Datacenters are rarely deployed to maximal capacity in one shot; rather, they are gradually expanded as network capacity demands increase.

4.1 The Practice of Expansion

In-place Expansion. At a high-level, expanding a topology involves two conceptual phases: (a) procuring new switches, servers, and cables and laying them on the datacenter floor, and (b) re-wiring (or adding) links between switches in the existing topology and the new switches. Phase (b), the *re-wiring phase*, can potentially disrupt traffic; as links are re-wired, network capacity can drop, leading to traffic loss. To avoid traffic loss, providers can either take the existing topology offline (migrate services away, for example, to another datacenter), or can carefully schedule link re-wiring while carrying live traffic, but schedule the re-wiring to maintain a desired target capacity. The first choice can impact service availability significantly.

So, today, datacenters are expanded while carrying live traffic [30, 12, 31, 38]. To do this, expansion is carried out in *steps*, where at each step, the capacity of the topology is guaranteed to be at least a percentage p of the capacity of the existing topology. This fraction is sometimes called the expansion SLO. Today, many providers operate at expansion SLOs of 75% [38]; higher SLOs of 85-90% can impact availability budgets less while allowing providers to carry more traffic during expansion.

The unit of expansion. Since expansion involves procurement, topologies are usually expanded in discrete units called *blocks* to simplify the procurement and layout logistics. In a structured topology, there are natural candidates for blocks. For example, in a Clos, a pod can be block, while in an Xpander, the metanode can be a block. During expansion, a block is first fully assembled and placed, and links between switches *within* a block are connected (as an aside, an Xpander metanode has no such links). During the re-wiring phase, *only links between existing blocks and new blocks are re-wired*. (This phase does *not* re-wire links between switches within an existing block). Aside from simplifying logistics, expanding at the granularity of a block preserves structure in structured topologies.

4.2 An Expansion Step

What happens during a step. Figure 2 shows an example of Clos expansion. The upper left figure shows a partially-deployed logical Clos, in which each spine and aggregation

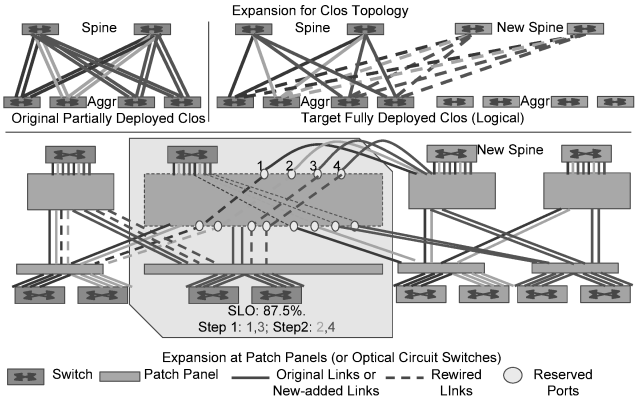


Figure 2: Clos Expansion with Patch Panels

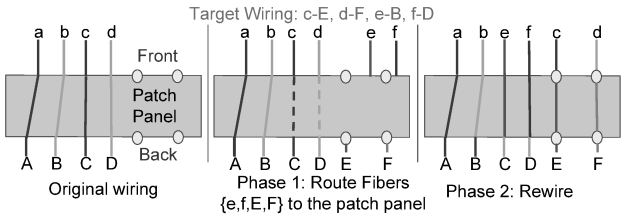


Figure 3: Basic Rewiring Operations at a patch panel

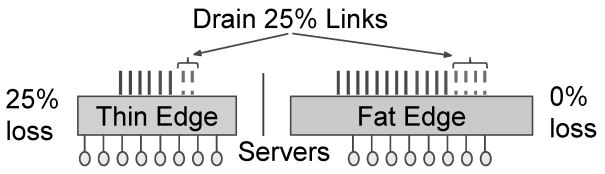


Figure 4: Thin and Fat Edge Comparison

block are connected by two links. The upper right is the target fully-deployed Clos, where each spine and aggregation block are connected by a single link. During expansion, we need to redistribute half of existing links (dashed) to the newly added spines without violating wiring and capacity constraints.

Suppose we want to maintain 87.5% of the capacity of the topology (*i.e.,* the expansion SLO is 0.875), this expansion will require 4 steps in total, where each patch panel is involved in 2 of these steps. In Figure 2, we only show the rewiring process on the second existing patch panels. To maintain 87.5% capacity at each pod, only one link is allowed to be drained. In the first step, the red link from the first existing aggregation block and the green link from the second existing aggregation block are rewired to the first new spine block. In the second step, the orange links from the first existing aggregation block and the purple link from the second existing aggregation block are rewired to the first new spine block. A similar process happens in the first patch panel.

In practice, each step of expansion involves four sub-steps. In the first sub-step, the existing links that are to be re-wired are *drained*. Draining a link involves programming switches at each end of the link to disable the corresponding ports, and may also require reprogramming other switches or ports to route traffic around the disabled link. Second, one or more human operators *physically rewire* the links at a patch panel (explained in more detail below). Third, the newly wired links are *tested* for bit errors by sending test traffic through them. Finally, the new links are *undrained*.

By far the most time consuming part of each step is the second sub-step, which requires human involvement. This sub-step is also the most important from an availability perspective; the longer this sub-step takes, the longer the datacenter operates at reduced capacity, which can impact availability targets [12].

The role of patch panels in re-wiring. The lower figure in Figure 2 depicts the physical realization of the (logical) re-wiring shown in the upper figure. (For simplicity, the figure only shows the re-wiring of links on one patch panel to a new pod). Fibers and bundles originate and terminate at patch panels, so re-wiring requires reconnecting input and output ports at each patch panel. One important constraint in this process is that re-wiring cannot remove fibers that are already part of an existing bundle.

Patch panels help localize rewiring and reuse existing cable bundling during expansions. Figure 3 shows, in more detail the rewiring process at a single patch panel. The leftmost figure shows the original wiring with connections (a, A), (b, B), (c, C), (d, D). To enable expansion, a topology is always deployed such that some ports at the patch panel are reserved for expansion steps. In the figure, we use these reserved ports to connect new fibers e, f, E and F (Phase 1). To get to a target wiring in the expanded network with connections (a, A), (b, B), (e, C), (f, D), (c, E), (d, F), the following steps are taken: (1) Traffic is drained from (c, C), (d, D), (2) Connections (c, C), (d, D) are rewired, with c being connected to E, d being connected to F and so on, and (3) The new links are undrained, allowing traffic to use new capacity.

4.3 Expansion Complexity Metrics

We identify two metrics that quantify expansion complexity and use these metrics to identify differences between Clos and Jellyfish in the next subsection.

Number of Expansion Steps. As mentioned each expansion step requires a series of substeps which cannot be parallelized. Therefore the number of expansion steps determines the total time for expansion.

Average number of rewired links in a patch panel rack per step. With patch panels, manual rewiring dominates the time taken within each expansion step. Within steps, it is possible to parallelize rewiring across racks of patch panels. With such parallelization, the time taken to rewire a single patch panel rack will dominate the time taken for each expansion step.

4.4 Comparing Topology Classes

Table 2 shows the value of these measures for a medium-sized Clos and a comparable Jellyfish topology, when the expansion

Topology	4-layer Clos (Medium)	Jellyfish
Average # links rewired per patch panel rack	832	470
Expansion steps	6	3
North-to-south capacity ratio	1	3

Table 2: Expansion Comparison (SLO = 90%)

SLO is 90%. (§6 has more extensive comparisons for these metrics, and also describes the methodology more carefully). In this setting, the number of links rewired per patch panel can be a factor of two less than Clos. Moreover, Jellyfish requires 3 steps, while Clos twice the number of steps.

To understand why Jellyfish requires fewer steps, we define a metric called the *north-to-south* capacity ratio for a block. This is the ratio of the aggregate capacity of all "northbound" links exiting a block to the aggregate capacity of all "southbound" links to/from the servers within the block. Figure 4 illustrates this ratio: a *thin edge* (left), has an equal number of southbound and northbound links while a *fat edge* (right), has more northbound links than southbound links. A Clos topology has a thin edge, *i.e.,* this ratio is 1, since the block is a pod. Now, consider an expansion SLO of 75%. This means that the southbound aggregate capacity must be at least 75%. That implies that, for Clos, *at most* 25% of the links can be rewired in a single step. However, Jellyfish has a much higher ratio of 3, *i.e.,* it has a fat edge. This means that *many more links can be rewired in a single step* in Jellyfish than in Clos. This property of Jellyfish is required for reducing the number of expansion steps.

Takeaway. Clos topologies re-wire more links in each patch panel during an expansion step and require many steps because they have a low north-south capacity ratio.

5 Towards Lower Lifecycle Complexity

Our discussions in §3 and §4, together with preliminary results presented in those sections (§6 has more extensive results) suggest the following qualitative comparison between Clos and the expander graph families with respect to lifecycle management costs (Table 3):

- Clos uses fewer bundle types and patch panels.
- Jellyfish has significantly lower switch counts, uses fewer expansion steps, and touches fewer links per patch panel during an expansion step.

In all of these comparisons, we compare topologies with the same number of servers and the same bisection bandwidth.

The question we ask in this paper is: *Is there a family of topologies which are comparable to, or dominate, both Clos and expander graphs by all our lifecycle management metrics?* In this section, we present the design of the FatClique class of topologies and validate in §6 that FatClique answers this question affirmatively.

5.1 FatClique Construction

FatClique (Figure 5) combines the hierarchical structure in Clos with the edge expansion in expander graphs to achieve lower lifecycle management complexity. FatClique has three

	4-layer Clos (Medium)	Jellyfish
switches		✓
bundle types	✓	
patch panels	✓	
re-wired links per patch panel		✓
expansion steps		✓

Table 3: Qualitative comparison of lifecycle management complexity

Auxiliary Variable	Description
$p_s = S_c - 1$	# ports per switch to connect other switches inside a sub-block
$p_b = k - s - p_s - p_c$	# ports per switch to connect other blocks
$R_c = S_c \cdot (p_c + p_b)$	radix of a sub-block
$R_b = S_b \cdot S_c \cdot p_b$	radix of a block
$N_b = N/(S_b \cdot S_c \cdot s)$	#blocks
$L_{cc} = S_c \cdot p_c/(S_b - 1)$	#links between two sub-blocks inside a block
$L_{bb} = R_b/(N_b - 1)$	#links between two blocks

Table 4: FatClique Variables

levels of hierarchy: individual sub-block (top left), interconnected into a block (top right), which are in turn interconnected to form FatClique (bottom). The interconnection used at every level in the hierarchy is a clique, similar to Dragonfly [20]. Additionally, *each level* in the hierarchy is designed to have a fat edge (a north-south capacity ratio greater than 1). The cliques enable high edge expansion, while hierarchy enables lower wiring complexity than random-graph based expanders [32, 35].

FatClique is a class of topologies. To obtain an instance of this class, a topology designer specifies two input parameters: N, the number of servers, and k the chip radix. A *synthesis* algorithm takes these as inputs, and attempts to *instantiate* four *design variables* that completely determine the FatClique instance Table 4. These four design variables are:

- s, the number of ports in a switch that connect to servers
- p_c, the number of ports in each switch that connect to other sub-blocks inside a block
- S_c, the number of switches in a sub-block
- S_b, the number of sub-blocks in a block

The synthesis algorithm searches for the best combination of values for design variables, guided by six constraints, C_1 through C_6, described below. The algorithm also defines *auxiliary* variables for convenience; these can be derived from the design variables (Table 4). We define these variables in the narrative below.

Sub-block connectivity. In FatClique, the sub-block forms the lowest level of the hierarchy, and contains switches and servers. All sub-blocks have the same structure. Servers are distributed uniformly among all switches of the topology, such that each sub-block has the same number of servers attached. However, because this number of servers may not be an exact multiple of the number of switches, we distribute the remainder across the switches, so that some switches may be connected to one more server than others. The alternative would have been to truncate or round up the number of servers per sub-block to be divisible by the number of switches in

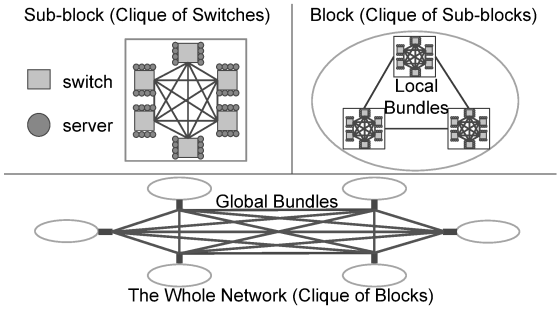


Figure 5: FatClique Topology

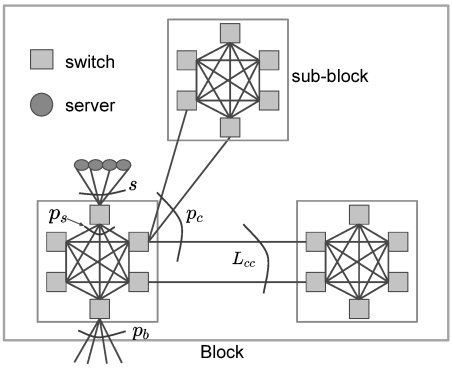


Figure 6: FatClique Block

the sub-block, which could lead to overprovisioning or underprovisioning. Within a sub-block, every switch has a link to every other switch within its sub-block, to form a clique (or complete graph). To ensure a fat edge at the sub-block level, each switch must connect to more switches than servers, captured by the constraint $C_1 : s < r - s$, where r is the switch radix and s is the number of ports on a switch connected to servers.

Block-level connectivity. The next level in the hierarchy is the block. Each sub-block is connected to other sub-blocks within a block using a clique (Figure 5, top-left). In this clique, each sub-block may have *multiple* links to another sub-block; these inter-sub-block links are evenly distributed among all switches in the sub-block such that every pair of switches from different sub-block has at most one link. Ensuring a fat edge at this level requires that a sub-block has more inter-sub-block and inter-block links egressing from the sub-block than the number of servers it connects to. Because sub-blocks contain switches which are homogeneous[5], this constraint is ensured if the sum of (a) the number ports on each switch connected to other sub-block (p_c) and (b) those connected to other blocks (p_b, an auxiliary variable in Table 4, see also Figure 6) exceeds the number of servers connected to the switch (captured by $C_2 : p_c + p_b > s$).

Inter-block connectivity. The top of the hierarchy is the overall network, in which each block is connected to every other block, resulting in a clique. The inter-block links are evenly distributed among all sub-blocks, and, within a sub-block, evenly among all switches. To ensure a fat edge at this level, the number of inter-block links at each switch should be larger than the number of servers it connects to, captured by $C_3 : p_b > s$. Note that C_3 subsumes (is a stronger constraint than) C_2. Moreover, the constraint that blocks are connected in a clique imposes a constraint on the *block radix* (R_b, a derived variable). The block radix is the total number of links in a block destined to other blocks. R_b should be large enough to reach all other blocks (captured by $C_4 : R_b \geq N_b - 1$) such that the whole topology is a clique.

Incorporating rack space constraints. Beyond connectivity constraints, we need to consider packaging constraints in sub-block design. Ideally, we need to ensure that a sub-block fits completely into one or more racks with *no wasted rack space*. For example, if we use 58RU racks, and each switch is to be connected to 8 1RU servers, we can accommodate 6 switches per sub-block, leaving $58 - (6 \times 8 + 6) = 4$U in the rack for power supply and other equipment. In contrast, choosing 8 switches per sub-block would be a bad choice because it would need $8 \times 8 + 8 = 72$U rack space, overflowing into a second rack that would have 44RU un-utilized. We model this packaging fragmentation as a *soft* constraint: our synthesis algorithm generates multiple candidate assignments to the design variables that satisfy our constraints, and of these, we pick the alternative that has the lowest wasted rack space.

Ensuring edge expansion. At each level of the hierarchy, edge expansion is ensured by using a clique. This is necessary for high edge expansion, but not sufficient, since it does not guarantee that every switch connects to as many other switches across the network as possible. One way to ensure this diversity is to make sure that each pair of switches is connected by at most one link. The constraints discussed so far do not ensure this. For instance, consider Figure 6, in which L_{cc} (another auxiliary variable in Table 4) is the number of links from one sub-block to another. If this number is greater than the number of switches S_c in the sub-block, then, some pair of switches might have more than one link to each other. Thus, $C_5 : L_{cc} \leq S_c$ is a condition to ensure that each pair of switches must be connected by a single link. Our topology synthesis algorithm generates assignments to design variables, and a topology generator then assigns links to ensure this property (§5.2).

Incorporating patch panel constraints. The size of the block is also limited by the number of ports in a single patch panel rack (denoted by $PP - Rack_{ports}$). It is desirable to ensure that the inter-block links egressing each block connect to at most $\frac{1}{2}$ the ports in a patch panel rack, so that the rest of the patch panel ports are available for external connections into the block (captured by $C_6 : R_b \leq \frac{1}{2} \cdot PP - Rack_{ports}$).

[5]They are nearly homogeneous, since a switch may differ from another by one in the number of servers connected

Topology	Scalability
3-layer Clos (Fattree)	$2 \cdot (k/2)^3$
4-layer Clos	$2 \cdot (k/2)^4$
5-layer Clos (Jupiter)	$2 \cdot (k/2)^5$
FatClique	$O(k^5)$

Table 5: Scalability of Topologies

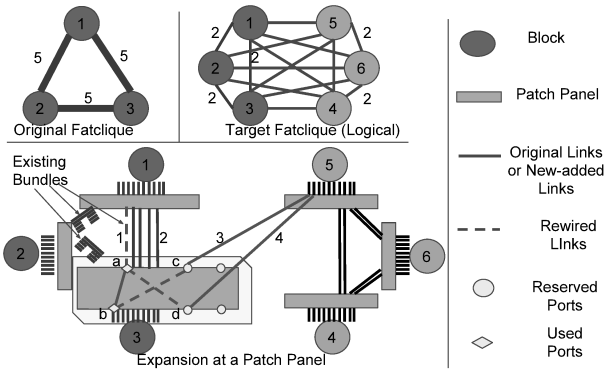


Figure 7: FatClique Expansion example

5.2 FatClique Synthesis Algorithm

Generating candidate assignments. The FatClique synthesis algorithm attempts to assign values to the design variables, subject to constraints C_1 to C_6. The algorithm enumerates all possible combinations of value assignments for these variables, and filters out each assignment that fails to satisfy all the constraints. For each remaining assignment, it generates the topology specified by the design variable, and determines if the topology satisfies a required capacity Cap^*, which is an input to the algorithm. Each assignment that fails the capacity test is also filtered out, leaving a candidate set of assignments. These steps are described in §A.6.

FatClique placement. For each assignment in this candidate set, the synthesis algorithm generates a *topology placement*. Because FatClique's design is regular, its topology placement algorithm is conceptually simple. A sub-block may span one or more racks, and these racks are placed adjacent to each other. All sub-blocks within a block are arranged in a rectangular fashion on the datacenter floor. For example, if a block has 25 racks, it is arranged in a 5×5 pattern of racks. Blocks are then arranged in a similar grid-like fashion.

Selecting best candidate. For each placement, the synthesizer computes the cabling cost of the resulting placement (using [7]), and picks the candidate with the lowest cost. This step is not shown in Algorithm 3. This approach implicitly filters out candidates whose sub-block cannot be efficiently packed into racks (§5.1).

5.3 FatClique Expansion

Re-wiring during expansion. Consider a small FatClique topology, shown top left in Figure 7, that has 3 blocks and $L_{bb} = 5$, *i.e.,* five inter-block links. To expand it to a clique with six blocks, we would need to rewire the topology to have $L'_{bb} = 2$ (top right in Figure 7). This means we need to redistribute more than half (6 out of 10) of existing links (red) at each block to new blocks without violating wiring and capacity constraints.

The expansion process with patch panels is shown in the bottom of Figure 7. Similar to the procedure for Clos described in §4.1, all new blocks (shown in orange) are first deployed and interconnected and links from the new blocks are routed to reserved ports on patch panels associated with existing blocks (shown in blue), before re-wiring begins.

For FatClique, rewiring one existing link requires releasing one patch panel port so that a new link can be added. Since links are already parts of existing bundles and routed through cable trays, we can not rewire them directly, *e.g.,* by rerouting it from one patch panel to another. For example, link 1 (lower half of Figure 7) is originally connected blocks 1 and 3 by connecting ports a and b on the patch panel. Suppose we want to remove that link, and add two links, one from block 1 to block 5 (labeled 3), and another from block 3 to block 5 (labeled 4). The part of the original link (labeled 1) between the two patch panels is already bundled, so we cannot physically reroute it from block 3 to block 5. Instead, we effect re-wiring by releasing port a, connecting link 3 to port a, connecting link 1 to port c. Logically, this is equivalent to connecting ports a and d and b and c on the patch panel shown in lower half of Figure 7. This preserves bundling, while permitting expansion.

If the original topology has N_b blocks, by comparing the old and target topology, the total number of rewired links is computed by $N_b(N_b - 1)(L_{bb} - L'_{bb})/2$. For this example, the total number of links to be rewired is 9.

Iterative Expansion Plan Generation. By design, FatClique has fat edges, which allows draining more and more links at each step of the expansion, as network capacity increases. At each step, we drain links across all blocks uniformly, so that each block loses the same aggregate capacity. However the relationship between overall network capacity, and the number of links drained at every block in FatClique is unclear, because traffic needs to be sent over non-shortest paths to fully utilize the fabric.

Therefore, we use an iterative approach to expansion planning, where, at each step, we search for the maximal ratio of links to be drained that still preserves expansion SLO. (§A.4 discusses the algorithm in more detail). Our evaluation §6 shows that the number of expansion steps computed by this algorithm is much smaller than that for expanding symmetric Clos.

5.4 Discussion

Achieving low complexity. By construction, FatClique achieves low lifecycle management complexity (Table 3), while ensuring full-bisection bandwidth. It ensures high edge expansion, resulting in fewer switches. By packaging clique connections into a sub-block, it exports fewer external ports, an idea we call *port hiding*. By employing hierarchy and a regular (non-random) structure, it permits bundling and re-

quires fewer patch panels. By ensuring fat edges at each level of the hierarchy, it enables fewer re-wired links per patch panel, and fewer expansion steps. We quantify these in §6.

Scalability. Since Xpander and Jellyfish do not incorporate hierarchy, they can be scaled to arbitrarily large sizes. However, because Clos and FatClique are hierarchical, they can only scale to a fixed size for a given chip radix. Table 5 shows the maximum scale of each topology as a function of switch radix k. FatClique scales to the same order of magnitude as a 5-layer Clos. As shown in §6, both of them can scale to 64 times bisection bandwidth of Jupiter.

FatClique and Dragonfly. FatClique is inspired by Dragonfly [20] and they are both hierarchical topologies that use cliques as building blocks, but differ in several respects. First, for a given switch radix, FatClique can scale to larger topologies than Dragonfly because it incorporates one additional layer of hierarchy. Second, the Dragonfly class of topologies is defined by many more degrees of freedom than FatClique, so instantiating an instance of Dragonfly can require an expensive search [33]. In contrast, FatClique's constraints enable more efficient search for candidate topologies. Finally, since Dragonfly does not explicitly incorporate constraints for expansion, a given instance of Dragonfly may not end up with fat edges.

Routing and Load Balancing on FatClique. Unlike for Clos, ECMP-based forwarding cannot be used achieve high utilization in more recently proposed topologies [20, 35, 32, 19]. FatClique belongs to this latter class, for which a combination of ECMP and Valiant Load Balancing [37] has been shown to achieve performance comparable to Clos [19].

6 Evaluating Lifecycle Complexity

In this section, we compare three classes of topologies, Clos, expander graphs and FatClique by our complexity metrics.

6.1 Methodology

Topology scales. Because the lifecycle complexity of topology classes can be a function of topology scale, we evaluate complexity across three different topology sizes based on the number of servers they support: *small*, *medium*, and *large*. Small topologies support as many servers as a 3-layer clos topology. Medium topologies support as many servers as 4-layer Clos. Large topologies support as many servers as 5-layer Clos topologies[6]. All our experiments in this section are based on comparing topologies at the same scale.

At each scale, we generate one topology for each of Clos, Xpander, Jellyfish, and FatClique. The characteristics of these topologies are listed in Table 6. All these topologies use 32-port switching chips, the most common switch radix available today for all port capacities [5]. To compare topologies fairly, we need to equalize them first. Specifically, at a given scale, each topology has approximately the same bisection bandwidth, computed (following prior work [32, 35]) using METIS [18]. All topologies at the same scale support roughly the same number of servers; small, medium and large scale topologies achieve, respectively, $\frac{1}{4}$, 4, and 16 times capacity of Jupiter. (In A.8, we also compare these topologies using two other metrics).

Table 6 also shows the scale of individual building blocks of these topologies in terms of number of switches. For Clos, we use the algorithm in §A.1 to design building blocks (chassis) and then use them to compose Clos. One interesting aspect of this table is that, at the 3 scales we consider, a FatClique's sub-block and block designs are *identical*, suggesting lower manufacturing and assembly complexity. We plan to explore this dimension in future work.

For each topology we compute the metrics listed in Table 3: the number of switches, the number of bundle types, the number of patch panels, the average number of re-wired links at a patch panel during each expansion step, and the number of expansion steps. To compute these, we need component parameters, and placement and expansion algorithms for each topology class.

Component Parameters. In keeping with [4, 40], we use optical links for all inter-rack links. We use 96 port 1RU patch panels [10] in our analysis. A 58RU [28] rack with patch panels can aggregate $2*96*58 = 11,136$ fibers. We call this rack a patch-panel rack. Most datacenter settings, such as rack dimensions, aisle dimensions, cable routing and distance between cable trays follow practices in [26]. We list all parameters used in our paper in §A.7.

Placement Algorithms. For Clos, following Facebook's fb-fabric [3], spine blocks are placed at the center of the datacenter, which might take multiple rows of racks, and pods are placed at two sides of spine blocks. Each pod is organized into a rectangular area with aggregation blocks placed in the middle to reduce the cable length from ToR to aggregation. FatClique's placement algorithm is discussed in §5.2. For Xpander, we use the placement algorithm proposed in [19]. We follow the practice that all switches in a metanode are placed closed to each other. However, instead of placing a metanode into a row of racks, we place a metanode into a rectangular area of racks, which reduces cable lengths when metanodes are large. For Jellyfish, we design a random search algorithm to aggressively reduce the cable length (§A.2).

Expansion Algorithms. For Clos, as shown in [38], it is fairly complex to compute the optimal number of rewired links for asymmetric Clos during expansion. However, when the original and target topologies are both symmetric, this number is easy to compute. For this case, we design an optimal algorithm (§A.5) which rewires the maximum number of links at each step and therefore uses the smallest number of steps to finish expansion. For FatClique, we use the algorithm discussed in §5.3. For Xpander and Jellyfish, we design an

[6]To achieve low wiring complexity, a full 5-layer Clos topology would require patch panel racks with four times as many ports as available today, so we restrict ourselves to the largest Clos that can be constructed with today's patch panel capacities

Topology	Clos						FatClique				Xpander			Jellyfish	
Scale	e	a	sp	pod	cap	svr	sub-block	block	cap	svr	metanode	cap	svr	cap	svr
Small	1	16	1	32	327T	8.2k	6	150	337T	8.1k	41	351T	8.2k	350T	8.2k
Medium	1	16	48	32	5.24P	131k	6	150	5.40P	132k	655	5.56P	131k	5.56P	131k
Large	1	512	48	768	20.96P	524k	6	150	21.36P	523k	2620	22.27P	524k	22.27P	524k

Table 6: Capacities of topologies built with 32 port 40G switches. Small, medium and large scale topologies achieve $\frac{1}{4}$, 4, 16 times capacity of Jupiter. The table also shows sizes of individual building blocks of these topologies in terms of number of switches. Abbreviations: e:edge, a:aggregation, sp:spine, cap:capacity, svr:server.

expansion algorithm based on the intuition from [35, 32] that, to expand a topology by n ports requires breaking $\frac{n}{2}$ existing links. Finally, we have found that for all topologies, the number of expansion steps at a given SLO is *scale invariant*: it does not depend on the size of the original topology as long as the expansion ratio (target-topology-size-to-original-topology-size ratio) is fixed (§A.3).

Presenting results. In order to bring out the relative merits of topologies, and trends of how cost and complexity increase with scale, we present values for metrics we measure for all topologies and scales in the same graph. In most cases, we present the absolute values of these metrics; in some cases though, because our three topologies span a large size range, for some metrics the results across topologies are so far apart that we are unable to do so without loss of information. In these cases, we normalize our results by the most expensive, or complex topology.

6.2 Patch Panel Placement

The placement of patch panels is determined both by the structure of the topology and its scale.

Between edge and aggregation layers in Clos. For small and medium scale Clos, no patch panels are needed between edge and aggregation layers. Each pod at these scales contains 16 aggregation switches, which can be packed into a single rack (we call this an *aggregation-rack*). Given that a pod at this scale is small, all links from the edge can connect to this rack. Since all links connect to one physical location, bundles form naturally. In this case, each bundle from edge racks contains 3×16 fibers[7]. Therefore, no patch panels are needed between edge and aggregation layers.

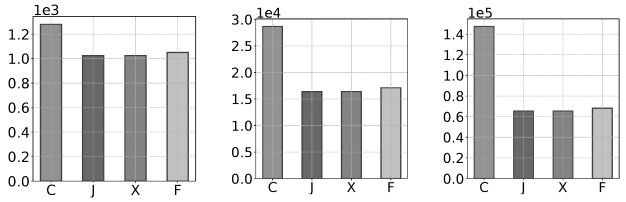
However, a large Clos needs one layer of patch panels between edge and aggregation layers since a pod at this scale is large. An aggregation block consists of 16 middle blocks[8], each with 32 switches. The aggregation block by itself occupies a single rack. Based on the logical connectivity, links from any edge need to connect to all middle blocks. Without using patch panels, each bundle could at most contain $3 \times 16/16 = 3$ fibers. In our design, we use patch panels to aggregate local bundles from edges first and then rebundle them on patch panels to form new high capacity bundles from patch panels to aggregation racks. Based on the patch panel rack capacity constraint, two patch panel racks are enough to form high capacity bundles from edge to aggregation layers. Specifically, in our design 128 edge switches and 8 aggregation racks connect to a single patch panel. In this design, each edge-side bundle contains 48 fibers and each aggregation-side bundle contains 128 fibers.

Between aggregation and spine layers. The topology between aggregation and spine layer in Clos is much larger than that inside a pod. For this reason, to form high capacity bundles, two layers of patch panels are needed. As shown in Figure 1, one layer of patch panels is placed near spine blocks at the center of the data center floor. Each patch panel rack aggregates local bundles from four spine racks in medium and large scale topologies. Similarly, another layer of patch panels are placed near aggregation rack, permitting long bundles between those patch panels.

In expanders and FatClique. For Jellyfish, Xpander and FatClique, patch panels are deployed at the server block side and long bundles form between those patch panels. In FatClique, each block requires one patch panel rack (§5.3). In a large Xpander, since a metanode is too big (Table 6), it is not possible to use one patch panel rack to aggregate all links from a metanode. Therefore, we divide a metanode into homogeneous sections, called sub-metanodes, such that links from a sub-metanode can be aggregated at one patch panel rack. For Jellyfish, we partition the topology into groups, each of which contains the same number of switches as in a block in FatClique, so each group needs one patch panel rack.

6.3 Deployment Complexity

In this section, we evaluate our different topologies by our three measures of deployment complexity (§3.2).



(a) Small (b) Medium (c) Large

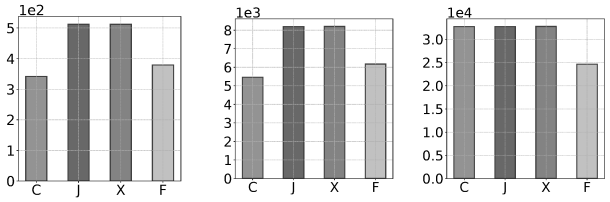
Figure 8: Number of switches. **C** is Clos, **J** is Jellyfish, **X** is Xpander and **F** is FatClique.

Number of Switches. Figure 8 shows how the different topologies compare in terms of number of switches used at various topology scales. Figure 8(a) shows the total number of

[7] In our setting, each rack with 58RU can accommodate at most 3 switches and 48 associated servers. The total number of links out of this rack is $3 * 16$.

[8] We follow the terminology in [31]. A middle block is a sub-block in an aggregation block.

switches for the small topologies, Figure 8(b) for the medium, and Figure 8(c) for the large. The y-axes increase in scale by about an order of magnitude from left to right. FatClique has 20% fewer switches than Clos for a small topology, and 50% fewer for the large. The results for Jellyfish and Xpander are similar, consistent with findings in [35, 32]. This benefit comes from the edge expansion property of the non-Clos topologies we consider. This implies that Clos topologies, at large scale, *may require nearly twice the capital expenditures* for switches, racks, and space as the other topologies.



(a) Small (b) Medium (c) Large

Figure 9: Number of patch panels. **C** is Clos, **J** is Jellyfish, **X** is Xpander and **F** is FatClique.

Number of Patch panels. Figure 9 shows the number of patch panels at different scales. As before, across these graphs, the y-axis scale increases approximately by one order of magnitude from left to right. At small and medium scales, Clos relies on patch panels mainly for connections between aggregation and spine blocks. Of all topologies at these scales, Clos uses the fewest number of patch panels: FatClique uses about 11% more patch panels, and Jellyfish and Xpander use almost 44-50% more. Xpander and Jellyfish rely on patch panels for all northbound links, and therefore in general, as scale increases, the number of patch panels in these networks grows (as seen by the increase in the y-axis scale from left to right).

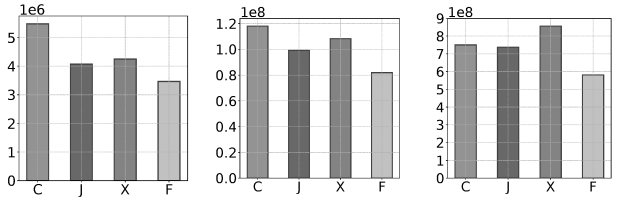
At large scale, however, Clos needs many more patch panels, comparable to Xpander and Jellyfish. At this scale, Clos aggregation blocks span multiple racks, and patch panels are also needed for connections between ToRs and aggregation blocks. Here, FatClique's careful packaging strategy becomes more evident, as it needs nearly 25% fewer patch panels than Clos. The majority of patch panels used in FatClique at all scales comes from inter-block links (which increase with scale).

For this metric, Clos and FatClique are comparable at small and medium scales, but FatClique dominates at large scale.

Scale	# Bundle Types			
	Clos	FatClique	Xpander	Jellyfish
Small	8	11	11	28
Medium	74	61	976	1577
Large	322	212	3034	3678

Table 7: Bundle Types (Switch Radix = 32)

Number of Bundle Types. Table 7 shows the number of bundle types used by different topologies at different scales. A bundle type (§3.1) is characterized by (a) the number of fibers in the bundle, and (b) the length of the bundle. The number of bundle types is a measure of wiring complexity. In this table, if bundles differ by more than 1m in length, they are designated as separate bundle types.



(a) Small (b) Medium (c) Large

Figure 10: Cabling cost. **C** is Clos, **J** is Jellyfish, **X** is Xpander and **F** is FatClique.

Table 7 shows that Clos and FatClique use the fewest number of bundle types; this is due to the hierarchical structure of the topology, where links between different elements in the hierarchy can be bundled. As the topology size increases, the number of bundle types also increases in these topologies, by a factor of about 40 for Clos to 20 for FatClique when going from small to large topologies.

On the other hand, Xpander and Jellyfish use an order of magnitude more bundle types compared to Clos and FatClique at medium and large scales, but use a comparable number for small scale topologies. Even at the small scale, Jellyfish uses many more bundle types because it uses a random connectivity pattern. At small scales Xpander metanodes use a single patch panel rack and bundles from all metanodes are uniform. With larger scales, Xpander metanodes become too big to connect to a single patch panel rack. We have to divide a metanode into several homogeneous sub-metanodes such that all links from sub-metanodes connect to a patch panel rack. However, because of the randomness in connectivity, this subdivision cannot ensure uniformity of bundles egressing sub-metanode patch panel racks, so we find that Xpander has a large number of bundle types in medium and large topologies.

Thus, by this metric, Clos and FatClique have the lowest complexity across all three scales, while Xpander and Jellyfish have an order of magnitude more complexity. Moreover, across all metrics FatClique has lowest deployment complexity, especially at large scales.

Case Study: Quantifying cabling costs. While not all aspects of lifecycle management complexity can be translated to actual dollar costs, it is possible to estimate one aspect, namely the cost of cables. Cabling cost includes the cost of transceivers and cables, and is reported to be the dominant component of overall datacenter network cost [31, 20]. We can estimate costs because our placement algorithms generate cable or bundle lengths, the topology packaging determines the number of transceivers, and estimates of cable and transceiver costs as a function of cable length are publicly available [7].

Figure 10 quantifies the cabling cost of all topologies,

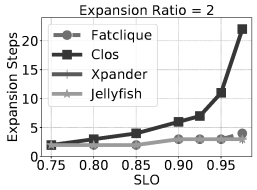


Figure 11: Expansion steps

across different scales. Clos has higher cabling costs at small and medium scales compared to expander graphs, although the relative difference decreases at medium scale. At large scales, the reverse is true. Clos is around 12% cheaper than Xpander in terms of cabling cost since Xpander does not support port-hiding at all and uses more long inter-rack cables. Thus, given that cabling cost is the dominant component of overall cost, it is unclear whether the tradeoff Xpander and Jellyfish makes in terms of number of switches and cabling design pays off in terms of capital expenditure, especially at large scale.

We find that FatClique has the lowest cabling cost of the topologies we study with a cabling cost 23-36% less than Clos. This result came as a surprise to us, because intuitively topologies that require all-to-all clique like connections might use longer length cables (and therefore more expensive transceivers). However on deeper examination, we found that Clos uses a larger number of cables (especially inter-rack cables) compared to other topologies since it has a relatively higher number of switches (Figure 8) to achieve the same bisection bandwidth. Thus, more switches leads to more racks and datacenter floor area, which stretches the cable length. All those factors together explain why Clos cabling costs are higher than FatClique's.

Thus, from an equipment capital expenditure perspective, at large scale *a FatClique can be at least 23% cheaper than a Clos*, because it has at least 23% fewer switches, 33% fewer patch panel racks, and 23% lower cabling costs than Clos.

6.4 Expansion Complexity

In this section, we evaluate topologies by our two measures of expansion complexity (§4.3): number of expansion steps required, and number of rewired-links per patch panel rack per step. Since the number of steps is scale-invariant (§6.1), we only present the results from expanding medium size topologies for both metrics[9]. When evaluating Clos, we study the expansion of symmetric Clos topologies; generic Clos expansion is studied in [38]. As discussed in §6.1, for symmetric Clos, we have developed an algorithm with optimal number of rewiring steps.

Number of expansion steps. Figure 11 shows the number of steps (y-axis) required to expand topologies to twice their existing size ($expansion\ ratio = 2$) at different expansion SLOs (x-axis). We find that at 75% SLO, all topologies require the same number of expansion steps. But the number

[9]We have verified that the relative trend in the number of re-wired links per patch panel holds for small and large topologies

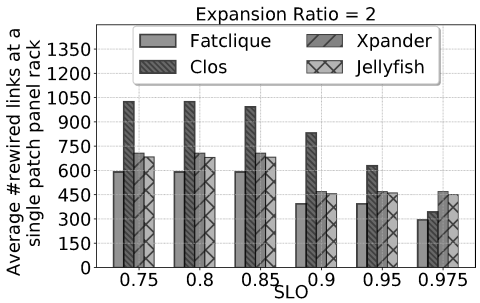


Figure 12: Average Number of Rewired Links at a Single Patch Panel across Steps

of steps required to expand Clos with tighter SLOs steeply increases. This is because the number of links that can be rewired per aggregation block in Clos per step, is limited (due to north-to-south capacity ratio §4.3) by the SLO. The tighter the SLO, fewer the number of links rewired per aggregation block per step, and larger the number of steps required to complete expansion. FatClique, Xpander and Jellyfish require fewer and comparable number of expansion steps due to their fat edge property, allowing many more links to be rewired per block per step. Their curves largely overlap (with FatClique taking one more step as SLO increases beyond 95%) .

Number of rewired links per patch panel rack per step. This metric is an indication of the time it takes to finish an expansion step because, today, rewiring each patch panel requires a human operator [38]. A datacenter operator can reduce re-wiring time by employing staff to rewire each patch panel rack in parallel, in which case, the number of links per patch panel rack per step is a good indicator of the complexity of an expansion step. Figure 12 shows the average of the maximum rewired links per patch panel rack, per step (y-axis), when expanding to twice the topology size size at different SLOs (y-axis). Even though the north-to-south capacity ratio restricts the number of links that can be rewired in Clos per step, the number of rewired links per patch panel rack per step in Clos remains consistently higher than other topologies, until we hit 97.5% SLO. The reason is that the links that need to be rewired in Clos are usually concentrated in few patch panel racks by design. As such, it is harder to parallelize rewiring in Clos, than it is in the other topologies. FatClique has the lowest rewiring step complexity across all topologies.

6.5 FatClique Result Summary

We find that FatClique is the best at most scales by *all* our complexity metrics. (The one exception is that at small and medium scales, Clos has slightly fewer patch panels). It uses 50% fewer switches and 33% fewer patch panels than Clos at large scale, and has a 23% lower cabling cost (an estimate we are able to derive from published cable prices). Finally, FatClique can permit fast expansion while degrading network capacity by small amounts (2.5-10%): at these levels, Clos can take 5 $\times$ longer to expand the topology, and each step of Clos expansion can take longer than FatClique because the

number of links to be rewired at each step per patch panel can be 30-50% higher.

7 Related Work

Topology Design. Previous topology designs have focused on cost effective, high capacity and low diameter datacenter topologies like [6, 35, 32, 4, 20]. Although they achieve good performance and cost properties, the lifecycle management complexity of these topologies have not been investigated either in the original papers or in subsequent work that has compared topologies [26, 27]. In contrast to these, we explore topology designs that have low lifecycle complexity. Recent work has explored datacenter topologies based on free space optics [24, 11, 9, 16, 39] but because we lack operational experience with them at scale, it is harder to design and evaluate lifecycle complexity metrics for them.

Topology Expansion. Prior work has discussed several aspects of topology expansion [30, 32, 35, 8, 38]. Condor [30] permits synthesis of Clos-based datacenter topologies with declarative constraints some of which can be used to specify expansion properties. A more recent paper [38] attempts to develop a target topology for expansion, given an existing Clos topology, that would require the least number of link rewiring. REWIRE [8] finds target expansion topologies with highest capacity and smallest latency without preserving topological structure. Jellyfish [32] and Xpander [35] study expansion properties of their topology, but do not consider practical details in re-wiring. Unlike these, our work is examines lifecycle management as a whole, across different topology classes, and develops new performance-equivalent topologies with better lifecycle management properties.

8 Conclusions and Future Work

In this paper, we have attempted to characterize the complexity of lifecycle management of datacenter topologies, an unexplored but critically important area of research. Lifecycle management consists of network deployment and expansion, and we devise metrics that capture the complexity of each. We use these to compare topology classes explored in the research literature: Clos and expander graphs. We find that each class has low complexity by some metrics, but high by others. However, our evaluation suggests topological features important for low lifecycle complexity: hierarchy, edge expansion and fat edges. We design a family of topologies called FatClique that incorporates these features, and this class has low complexity by all our metrics at large scale.

As the management complexity of networks increases, the importance of *designing for manageability* will increase in the coming years. Our paper is only a first step in this direction; several future directions remain.

Topology oversubscription. In our comparisons, we have only considered topologies with an over-subscription ratio of 1:1. Jupiter [31] permits over-subscription at the edge of the network, but there is anecdotal evidence that providers also over-subscribe at higher levels in Clos topologies. To explore the manageability of over-subscribed topologies it will be necessary to design over-subscription techniques in FatClique, Xpander and Jellyfish in a way in which all topologies can be compared on a equal footing.

Topology heterogeneity. In practice, topologies have a long lifetime over which they accrue heterogeneity: new blocks with higher radix switches, patch panels with different port counts *etc.* These complicate lifecycle management. To evaluate these, we need to develop data-driven models for how heterogeneity accrues in topologies over time and adapt our metrics for lifecycle complexity to accommodate heterogeneity.

Other management problems. Our paper focuses on topology lifecycle management, and explicitly does not consider other network management problems like fault isolation or control plane complexity. Designs for manageability must take these into account.

References

[1] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. Hyperx: Topology, routing, and packaging of efficient large-scale networks. In *Proc. SC9*, 2009.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. ACM SIGCOMM*, 2008.

[3] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[4] M. Besta and T. Hoefler. Slim fly: A cost effective low-diameter network topology. In *Proc. SC14*, 2014.

[5] Broadcom Inc. Broadcom Tomahawk Swiching chips. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series.

[6] C. Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, 32(2):406–424, March 1953.

[7] Colfax International. Colfax direct. http://www.colfaxdirect.com.

[8] Andrew R. Curtis, Tommy Carpenter, Mustafa Elsheikh, Alejandro López-Ortiz, and Srinivasan Keshav. Rewire: An optimization-based framework for unstructured data center network design. In *Proc. IEEE INFOCOMM*, 2012.

[9] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proc. ACM SIGCOMM*, 2010.

[10] FS.COM. 96 Fibers 12x MTP/MPO-8 to LC/UPC Single Mode 1U 40GB QSFP+ Breakout Patch Panel Flat. https://www.fs.com/products/43552.html.

[11] M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. Kilper. Projector: Agile reconfigurable data center interconnect. In *Proc. ACM SIGCOMM*, 2016.

[12] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proc. ACM SIGCOMM*, 2016.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *Proc. ACM SIGCOMM*, 2009.

[14] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proc. ACM SIGCOMM*, 2008.

[15] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *Proc. ACM SIGCOMM*, 2008.

[16] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proc. ACM SIGCOMM*, 2014.

[17] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc.*, 43(04):439–562, August 2006.

[18] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 1998.

[19] S. Kassing, A. Valadarsky, G. Shahaf, M. Schapira, and A. Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proc. ACM SIGCOMM*, 2017.

[20] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, 2008.

[21] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. Computers*, C-24(12):1145–1155, Dec 1975.

[22] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Proc. USENIX NSDI*, 2013.

[23] S. Mandal. Lessons learned from b4, google's sdn wan. https://www.usenix.org/sites/default/files/conference/protected-files/atc15_slides_mandal.pdf.

[24] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proc. ACM SIGCOMM*, 2017.

[25] J. Mitchell. What are Patch Panels & When to Use Them? https://www.lonestarracks.com/news/2016/10/28/patch-panels/.

[26] J. Mudigonda, P. Yalagandula, and J. C. Mogul. Taming the flying cable monster: A topology design and optimization framework for data-center networks. In *Proc. USENIX ATC*, 2011.

[27] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica. A cost comparison of datacenter network architectures. In *Proceedings of the 6th International COnference*, Co-NEXT '10, 2010.

[28] RackSolutions. Open Frame Server Racks. https://www.racksolutions.com/server-racks-cabinets-enclosures.html.

[29] Rackspace US, INC. The Rackspace Cloud. www.rackspacecloud.com.

[30] B. Schlinker, R. N. Mysore, S. Smith, J. C. Mogul, A. Vahdat, M. Yu, E. Katz-Bassett, and M. Rubin. Condor: Better topologies through declarative design. In *Proc. USENIX NSDI*, 2015.

[31] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proc. ACM SIGCOMM*, 2015.

[32] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *Proc. USENIX NSDI*, 2012.

[33] M. Y. Teh, J. J. Wilke, K. Bergman, and S. Rumley. Design space exploration of the dragonfly topology. In *ISC Workshops*, 2017.

[34] The Siemon Company. Trunk Cable Planning & Installation Guide. https://www.siemon.com/us/white_papers/07-09-24-trunk-cable-planning-installation.asp.

[35] A. Valadarsky, G. Shahaf, M. Dinitz, and M. Schapira. Xpander: Towards optimal-performance datacenters. In *Proc. ACM CoNEXT*, 2016.

[36] M. R. Zargham. *Computer Architecture: Single and Parallel Systems*. Prentice Hall, 1996.

[37] R. Zhang-Shen and N. McKeown. Designing a predictable internet backbone with valiant load-balancing. In *Proc. IEEE IWQoS*, 2005.

[38] S. Zhao, R. Wang, J. Zhou, J. Ong, J. Mogul, and A. Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *Proc. USENIX NSDI*, 2019.

[39] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. In *Proc. ACM SIGCOMM*, 2012.

[40] D. Zhuo, M. Ghobadi, R. Mahajan, K.-T. Förster, A. Krishnamurthy, and T. Anderson. Understanding and mitigating packet corruption in data center networks. In *Proc. ACM SIGCOMM*, 2017.

A Appendix

A.1 Clos Generation Algorithm

For Clos topologies, the canonical recursive algorithm in [36] can only generate non-modular topologies as shown in Figure 13. In practice, as shown in Jupiter [31], the topology is composed of heterogenous building blocks (chassis), which are packed into a single rack and therefore enforce port hiding (the idea that as few ports from a rack are exposed outside the rack). Although Jupiter is modular and supports port hiding, it is single instance of a Clos-like topology with a specific set of parameters. We seek an algorithm that can take any valid set of Clos parameters and produce chassis-based topologies automatically. Besides, it would be desirable for this algorithm to generate all possible feasible topologies satisfying the parameters, so we can select the one that is most compactly packed.

Our logical Clos generation algorithm achieves these goals. Specifically, the algorithm uses the following steps:

1. *Compute the total number of layers of homogeneous switching* chips needed. Namely, given N servers and radix k switches, we use $n = \log_{\frac{k}{2}}(\frac{N}{2})$ to compute the number of layers of chips n needed.

2. *Determine the total number of layers of chips for edge, aggregation and core layers*, which are represented by e, a and s respectively, such that $e + a + s = n$.

3. *Identify blocks for edge, aggregation and core layer.* Clos networks rely on every edge being able to reach every spine through exactly one path, by fanning out via as many different aggregation blocks as possible (and vice versa). We find that the resulting interconnection is a derivative of the classical perfect shuffle Omega network ([21], *e.g.,* aggregation blocks in Figure 14 and Figure 15). Therefore, we use Omega networks to build both the edge and aggregation blocks, and to define the connections between edge-aggregation and aggregation-spines. The spine block on the other hand needs to be rearrangeably-nonblocking, so it can relay flows from any edge to any other edge with full capacity. Therefore it is built as a smaller Clos topology [6] (*e.g.,* spine blocks in Figure 14).

4. *Compose the whole network using edge, aggregation and core blocks.* The process to compose the whole topology is to link all these blocks and uses the same procedure as Jupiter[31].

We have verified that topologies generated by our construction algorithm, such as the ones in Figure 14 and Figure 15, are isomorphic to a topology generated using the canonical algorithm in Figure 13. By changing different combinations of e, a and s, we can obtain multiple candidate topologies, as shown in Figure 14 and Figure 15.

A.2 Jellyfish Placement Algorithm

For Jellyfish, we use a heuristic random search algorithm to place switches and servers. The algorithm works as follows. At each stage of the algorithm, a node can be in one of two states: placed, or un-placed. A placed node is one which has been positioned in a rack. Each step of the algorithm randomly selects an un-placed node. If the selected node has logical neighbor nodes that have already been placed, we place this node at the centroid of the area formed by its placed logical neighbors. If no placed neighbor exists, the algorithm randomly selects a rack to place the node. We have also tried other heuristics like neighbor-first, which tries to place a switch's logical neighbors as close as possible around it. However, this performs worse than our algorithm.

A.3 Scale-invariance of Expansion

Scale-invariance of Expandability for Symmetric Clos. For a symmetric Clos network, the number of expansion steps is scale-invariant and independent of the degree to which the original topology is partially deployed. Consider a simplified Clos where the original topology has g aggregation blocks. Each aggregation block has p ports for spine-aggregation links, each of which has the unit capacity. Assume the worst-case traffic in which all sources are located in the left half of aggregation blocks and all destinations are in the right half. This network contains $g \cdot p/2$ crossing links between left and right halves. If, during expansion, the network is expected to support a demand of d units capacity per aggregation block, the total demand traversing the cut between the left and right halves in one direction is $d \cdot g/2$. Then, the maximum number of links that can be redistributed in an expansion step is $k = g \cdot p/2 - d \cdot g/2 = g(p-d)/2$, which is linear in the number of aggregation blocks (network size). This linearity between k and g implies scale-invariant expandability, *e.g.,* when an aggregation block is doubled to $2g$, the maximum number of redistributed links per expansion step becomes $2k$.

Scale-invariance of Expandability for Jellyfish, Xpander, and FatClique. A random graph consists of s nodes, which is a first-order approximation for Jellyfish's switch, Xpander's metanode and FatClique's block. Each node has p inter-node ports, so there are $s \cdot p/2$ inter-node links. We can treat the network as a bipartite graph. We assume the worst-case traffic matrix, where all traffic is sent through one part of the bipartite graph to the other. Suppose an expansion SLO requires each source-destination node pair to support d unit demand. Then the total demands from all sources are $d \cdot s/2$. The probability of a link being a cross link is $1/2$, and the expected number of cross links is $s \cdot p/4$. These cross links are expected to be the bottleneck between the source-destinations pairs. Therefore, in the first expansion step, we can redistribute at most $k = s \cdot p/4 - d \cdot s/2 = s(p/4 - d/2)$ links, and the maximum number of redistributed links is linear in the number of nodes (network size), *e.g.,* if the number of

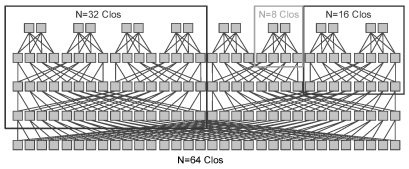


Figure 13: Recursive Construction

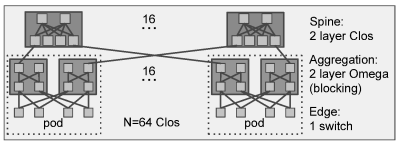


Figure 14: Block-Based Construction 1

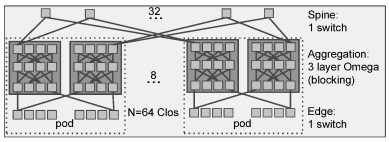


Figure 15: Block-Based Construction 2

nodes is doubled to $2s$, we can redistribute $2k$ links in the first step. It is easy to see that, after each expansion step, the number of links added to the bottleneck is also linear with the number of nodes, so the expandability is scale-invariant.

A.4 FatClique Expansion Algorithm

Algorithm 1 shows the expansion algorithm for FatClique. The input to the algorithm includes original and target topologies T^o and T^n, the link break ratio during an expansion step α, multipliers $\beta < 1$ and $\gamma > 1$, which are used to adjust α based on network capacity. α specifies the fraction of existing links that must be broken for re-wiring. The output of the algorithm is the expansion plan $Plan$.

Our expansion algorithm is an iterative trial-and-error approach (Line 4). Each iteration tries to find the right amount of links to break while satisfying the aggregate capacity constraint (Line 11) and the *edge capacity constraint* (Line 6), which guarantees that the *north-to-south* capacity ratio is always not smaller than 1 during any expansion step. If all constraints are satisfied, we accept this plan and tentatively increase the link break ratio α (Line 16, by multiplying by γ) due to capacity increase. Otherwise, the link break ratio α (Line 12) is decreased (by multiplying by β conservatively.)

```
   input  : T^o, T^n, SLO
   output: Plan
1  Initialize α ∈ (0,∞), β ∈ (0,1), γ ∈ (1,∞)
2  Find the total set of links to break, L, based on T^o and T^n
3  Compute original capacity c_0
4  while |L| > 0 do
5      Select a subset of links L_b, from L uniformly across all
         blocks, where |L_b| = α|L|.
6      if L_b does not satisfy edge capacity constraint then
7          α = α · β
8      end
9      Delete L_b from T^o
10     c = ComputeCapacity(T^o)
11     if c < c_0 · SLO then
12         α = α · β
13         add L_b back to T^o
14     else
15         T^o = AddNewLinks(L_b, T^o, T^n)
16         α = α · γ
17         Plan.add(L_b)
18     end
19 end
```

Algorithm 1: FatClique Expansion Plan Generation

A.5 Expansion for Clos

Since the motivation of this work is to compare topologies, we only focus on developping optimal expansion solutions for symmetrical Clos. More general algorithms for Clos' expansion can be found in [38]. Also, similar to [38], we assume the worst case traffic matrices for Clos, *i.e.,* servers under a pod will send traffic using full capacity to servers in other pods.

Target Topology Generation. As mentioned in §4.1, a pod is the unit of expansion in Clos. When we add new pods and associated spines to a Clos topology for expansion, the wiring pattern inside a pod remains unchanged. To make the target topology non-blocking and to ease expansion (*i.e.,* number of to-be-redistributed links on each pod is the same), links from a pod should be distributed across all spines as evenly as possible.

Expansion plan generation. Once a target Clos topology is generated, the next step is to redistribute links to convert the original topology into the target topology. By comparing the original and target topology, it is easy to figure out which new links should be routed to which patch panels to satisfy the wiring constraint. In this section, we mainly focus on how to drain links such that the capacity constraint is satisfied and the number of expansion steps is minimized.

Insight 1: Maximum rewired links at each pod is bounded. At each expansion step, when links are drained, network capacity drops. At the same time, as expansion proceeds, new devices are added incrementally, the overall network capacity increases gradually during the whole expansion process. In general, during expansion, the incrementally added capacity should be leveraged to speed up the expansion process. Due to the thin edges in Clos, no matter what the overall network capacity is, the maximum number links to be drained at each pod is bounded by the number of links on each pod multiplied by $(1-SLO)$. Figure 16 shows an example. The leftmost figure is a folded Clos, where each pod has 16 links (4 trunks). If the SLO is 75%, the maximum number of links to be drained at a single step is $16 \times (1-0.75) = 4$. For our expansion plan generation algorithm, we try to achieve this bound at each pod at every single step.

Insight 2: Drain links at spines uniformly across edges (pods). Given the number of links allowed to be drained at each pod, we need to carefully select which links are to be drained. Figure 16 shows two draining plans. Drain plan 1 will drain links from two spines uniformly across all pods. The residual capacity is 48, satisfying the requirement

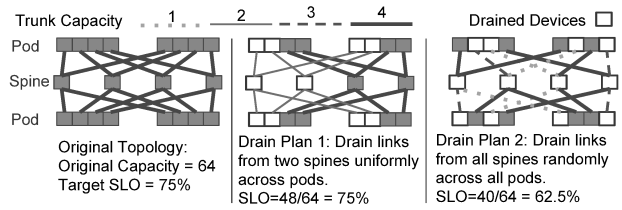


Figure 16: Original topology is a Folded Clos with capacity=64. The required SLO during expansion is 75%, which means capacity should be no smaller than 48. There are 16 links on each pod. Due to the SLO constraint, for all plans, 4 links are allowed to be drained at each pod.

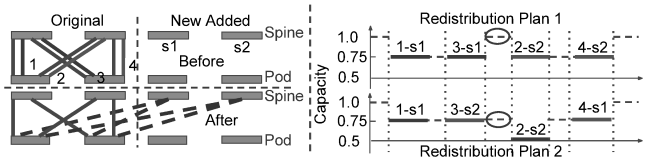


Figure 17: Clos Draining Link Redistribution Scheduling.

SLO=75%. By uniformly, we mean the number of drained links between the spine and all pods are the same. Drain plan 2 also drains 4 links from each pods but not uniformly (for example, more links are drained at the third spine compared to the fourth spine), which violates the SLO requirement since the residual capacity is only 40, smaller than the 48 in Drain plan 1.

Insight 3: Create physical loops by selecting the right target spines. Ideally, drained links with the same index on a pod on the same original spine should be redistributed to the same spine because the traffic sent from the pod to the target spine has a return path to the pod. Otherwise, the traffic will be dropped. Figure 17 illustrates this insight. The right side of the figure shows the performance of two redistribution plans. The y axis shows the normalized capacity of the network at each expansion step. In the first plan, link 1 is first moved to spine s1 (1-s1),followed by link 3 to the same spine s1 (3-s1) which results in 75% capacity loss, since the two pods are connected by three paths instead of four. Once links 1 and 3 are undrained, s1 connects the two pods by a fourth path, and the normalized capacity is restored to 1. This redistribution step now provides leeway for supporting 25% capacity loss in the next step. In this next step, links 2 and 4 are rewired to connect to s2. During the rewiring, capacity again drops to 75%, with three paths between the pods. On undraining links 2 and 4, the capacity is once again restored to 1. In contrast, redistribution plan 2 violates SLO because it does not focus on restoring capacity by establishing paths via the new spine, as suggested by the insight (links 1 and 3 are moved to different spines).

Inspired by these insights, we designed Algorithm 2, which can achieve all our insights simultaneously when both original and target topologies are symmetric. *The algorithm is optimal since at every expansion step, it achieves the upper bound of the links that could be drained. Therefore, our algorithm uses smallest steps to expand Clos.*

The input to the algorithm is the original and new *symmetric* topology T^o and T^n. We use T^o_{sp} and T^n_{sp} to represent the number of links between spine s and pod p in the old and new topology respectively. Initially, $T^o_{s'p} = 0$, where s' is a new spine. The output of the algorithm is the draining plan, $Subplan_i$, for expansion step i. The final expansion plan $Plan = \{Subplan_i\}$ and the number of $Subplan$, $|Plan|$, is the total expansion step.

The algorithm starts by indexing old spines, new spines and links on each pod from left to right respectively (Line 1-2), which are critical for the correctness of the algorithm since the algorithm relies on these indexes to break ties when selecting spines and links to redistribute. Then, based on our Insight 1, Line 3 computes the upper bound on the number of links to be redistributed on each pod, n_p. We show experimentally that our algorithm can always achieve this upper bound in each individual step as long as T^o and T^n are symmetric. Next, the algorithm iterates over all indexed old spines (Line 4) and tries to drain n_p links uniformly across all pods (Line 5) such that Insight 2 is satisfied. Line 6 compares the number of remaining to-be-redistributed links δ_{sp} and n_p and is useful only at the last expansion step. For each pod, the algorithm needs to find spines to redistribute links to (Line 7-14) while satisfying the constraint in Insight 3, *i.e.,* drained links with the same index on a pod on the same original spine are redistributed to the same spine. Due to indexing and symmetric structure of Clos, our algorithm can always satisfy Insight 3. Specifically, when selecting spines, the spine satisfying $\delta_{s'p} = T^n_{s'p} - T^o_{s'p} > 0$ with the smallest index will be considered first (Line 8-Line 10). When selecting links from pod to redistribute, we always select the first n_a links to redistribute (Line 14).

Theorem 1 *Algorithm 2 produces the optimal expansion plan for Clos topology.*

The proof is simple. Since at every expansion step, our algorithm achieves the upper bound of the links that could be drained, our algorithm uses smallest steps to finish the expansion.

A.6 FatClique Topology Synthesis Algorithm

The topology synthesis algorithm for FatClique is shown in Algorithm 3. Essentially, the algorithm is a search algorithm, and leverages the constraints C_1 to C_6 in §5.1 to prune the search space. It works as follows. The outermost loop (Line 2) enumerates the number of racks used for a sub-block. Based on the rack space constraints, sub-block size S_c is determined Line 4. Next, the algorithm iterates over the number of sub-blocks in a block S_b Line 5, whose size is constrained by $MaxBlockSize$. Inside this loop, we leverage constraints C_1 to C_6 and derivations in §5.1 to find the feasible set of p_c, which is represented by P_c (Line 6). Then we construct FatClique based all design variables Line 8 and compute its capacity Line 9. If the capacity matches the target capacity,

```
input : T^o, T^n, SLO
output: Subplan
Index original and new spines from left to right starting from 1 respectively
Index links at each pod from left to right starting from 1
∀ pod p, n_p = num_links_per_pod · (1-SLO)
                                                    // Insight 1
foreach Original Spine s do
    foreach pod p do                                // Insight 2
        δ_sp = T^o_sp − T^n_sp, n_p = min(n_p, δ_sp)
                                                    // Insight 2
        while n_p > 0 do
            foreach New Spine s' do                 // Insight 3
                δ_s'p = T^n_s'p − T^o_s'p
                if δ_s'p > 0 then break
            end
            n_a = min(δ_s'p, n_p)
            Find the first n_a to-be-distributed links, L_sp
            n_p = n_p − n_a, update(T^o)
            Subplan.add(L_sp)
        end
    end
end
```

Algorithm 2: Single Step Clos Expansion Plan Generation

we add this topology into candidate set (Line 15). If the capacity is larger than required, the algorithm will increase s by 1 which will decrease the number of switches used $n = N/s$ (N is fixed) and therefore reduce the network capacity in next search step (Line 13). If the capacity is smaller than required, the algorithm will decrease s by 1 (Line 11) to increase the number of switches and capacity in next search step.

A.7 Parameter Setting

The cable price with transceivers used in our evaluation is listed in Table 9. We found that a simple linear model does not fit the data. The data is better approximated by a piecewise linear function: cables shorter than 100 meters are fit using one linear model and cables beyond 100 meters are fit using another linear model. The latter has a larger slope because beyond 100 meters, more advanced and expensive transceivers are necessary. In our experiment, since we only know the discrete price for cables and associated transceivers, we do the following: if the length of the cable is X, we use the exact price; if the length if larger than X, we use the first cable price larger than X.

```
input : N,r,Cap*,s_0
output: candidate
candidate = []
for i = 1; i < MaxRackPerSubblock; i++ do
    s = s_0
    S_c = i · RackCapacity/(1+s)
    for S_b = 1; S_b <= MaxBlockSize; S_b++ do
        P_c = CheckConstraints(S_c, S_b)
        foreach p_c in P_c do
            T = ConstructTopology(S_c, S_b, s, p_c)
            Cap = ComputeCapacity(T)
            if Cap < Cap* then
                s = s − 1
            else if Cap > Cap* then
                s = s + 1
            else
                candidate.append(T)
            end
        end
    end
end
```

Algorithm 3: FatClique Topology Synthesis Algorithm

Rack width	24 inches
Rack depth	28.875 inches
Rack height	108 inches
Tray-to-rack distance	24 inches
Dist. Betw. cross-trays	48 inches
Aisle Width	48 inches
Rack units per rack	58 RU [29]
#Ports per patch panel	48 [10]
Patch panel space	1 RU
Cable tray size	24 inches x 4 inches [34]

Table 8: Datacenter settings mostly [26]

Length	3	5	10	15	20	30
Price	303	310	318	334	350	399
Length	50	100	200	300	400	
Price	489	753	1429	2095	2700	

Table 9: 40G QSFP Mellanox cable length in meter (Length) and price with transceivers (Price) [7]

A.8 Other Metrics

In our evaluations, we have tried to topologies with qualitatively similar properties 6. In this section, we quantify other properties of these topologies.

Edge Expansion and Spectral Gap. Since computing edge expansion is computationally hard, we follow the method in [35] using spectral gap [17] to approximate edge expansion. A larger spectral gap implies larger edge expansion. To fairly compare topologies, we equalize their bisection bandwidth first. As shown before, to achieve the same bisection

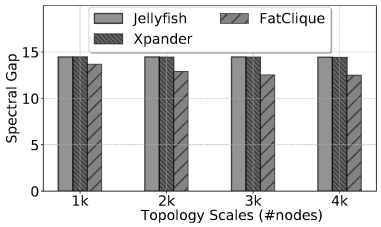


Figure 18: Spectral Gap

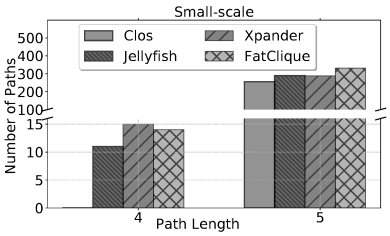


Figure 19: Path Diversity for Small-scale Topologies

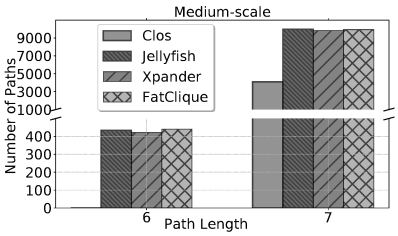


Figure 20: Path Diversity for Medium-scale Topologies

bandwidth, Clos uses many more switches. Also, Clos is not a d-regular graph and do not know of a way to compute the spectral graph for Clos-like topologies. Therefore, we compare the spectral gap only for d-regular graphs, Jellyfish, Xpander and FatClique at different scales (1k-4k nodes). The spectral gap is defined as follows [17]. Let G with node degree d and $A(G)$ denote the d-regular topology and its adjacent matrix. The matrix $A(G)$ has n real eigenvalues which we denote by $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. Spectral gap $SG = d - \lambda_2$. In our experiments, chip radix is 32 and each node in those topologies connects to 8 servers, $d = 24$. The result is shown in Figure 18. First, we observe that spectral gap stays roughly the same under different scales. Also, the spectral gap of FatClique is slightly lower than that of other topologies, which implies that FatClique has slightly smaller edge expansion compared to Jellyfish and Xpander. This is to be expected, since FatClique adds some hierarchical structure to cliques.

Path Diversity. We compute the path diversity for different topologies. For Clos, we only calculate the number of shortest paths between two ToR switches from different pods. For other topologies, we compute the number of paths which are no longer than the shortest paths in the same-scale Clos. For example, for small-scale Clos, the shortest path length is 5. We will only calculate paths whose length is no larger than 5 in other topologies. This is a rough metric for path diversity. The results are shown in Figure 19 and Figure 20. We found that Jellyfish, Xpander and FatClique have the same level of path diversity, which is higher than that of Clos. Also, those topologies have shorter paths than Clos.