USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Matryoshka: Realizing Hyperscale Data Center Network Design for the AI Era

Yan Cai, *Meta;* Jialong Li, *Max Planck Institute for Informatics;* Kutalmis Akpinar, Tianxiang Li, Hany Morsy, Jason Wilson, and Sunil Khaunte, *Meta;* Yiting Xia, *Max Planck Institute for Informatics;* Ying Zhang, *Meta*

https://www.usenix.org/conference/nsdi26/presentation/cai

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Matryoshka: Realizing Hyperscale Data Center Network Design for the AI Era

Yan Cai[1]*, *Jialong Li*[2]*†, *Kutalmis Akpinar*[1], *TianXiang Li*[1], *Hany Morsy*[1], *Jason Wilson*[1], *Sunil Khaunte*[1], *Yiting Xia*[3], *Ying Zhang*[1]

[1]*Meta* [2]*Shenzhen University of Advanced Technology* [3]*Max Planck Institute for Informatics*

Abstract

Over the past decade, data center networking (DCN) has undergone substantial transformation in terms of both scale and complexity. Developing a DCN entails multiple intricate steps, such as establishing physical connections, configuring logical network addressing, and defining high-level routing policies. While extensive work has focused on logical DCN design and physical deployment, a critical gap remains: materializing these designs into concrete switch configurations—a necessary step to realize the development procedure. This problem is especially acute in the AI era, as hyperscale, rapidly evolving, and highly heterogeneous AI-driven clusters place unprecedented demands on DCN design and implementation.

This paper presents Matryoshka, Meta's production-scale DCN design system that bridges this gap. Matryoshka employs an intent-based, model-driven approach to systematically compile high-level DCN design intents into working switch configurations. Operational for over six years, Matryoshka has supported orders-of-magnitude growth in Meta's DCN infrastructure, guiding the design nearly 900 DCNs across 18 distinct types, including the latest 100K-GPU supercluster for AI training. We share our experience in building and operating Matryoshka, highlighting how it empowers the rapid design and evolution of AI clusters nowadays.

1 Introduction

Data center (DC) networks (DCNs) form the foundation of modern cloud infrastructure, supporting a wide range of online services such as web search, storage, advertising, gaming, and video conferencing. As the bandwidth requirement doubles every 12–15 months [31], hyperscalers are investing heavily in DCN construction to meet the surging demand. The rise of Artificial Intelligence (AI) and Generative AI [11, 26] further accelerates this growth, as training large models and processing massive datasets require increasingly network-intensive operations at unprecedented scale.

Building hyperscale DCNs is a complex, multi-dimensional endeavor that must balance technical, operational, physical, and economic considerations—yet the process remains largely underexplored. Industry efforts have primarily focused on topology, both in terms of logical design [25, 30] and physical deployment through packaging, placement, and wiring [31, 38, 39]. However, the critical intermediate step—generating thousands of switch configurations that bridge logical topology and physical realization—is often overlooked. This includes interface naming, connectivity port mapping, IP address assignment, routing domain allocation, routing policies and configurations that all together bring the network to life. Academic research, by contrast, tends to concentrate on post-deployment configuration management [8, 10, 23], offering limited insight into how these configurations are generated in the first place.

In this paper, we fill this gap with Matryoshka, Meta's software tooling system for realizing DCN designs. Matryoshka adopts intent-based networking, compiling designer-specified high-level network intents into concrete switch configurations. Unlike conventional practices that rely on expertise-driven manual workflows (Method of Procedures) and scattered scripts for individual steps, Matryoshka provides an end-to-end, fully automated solution that delivers complete configurations as a unified package. Each Matryoshka operation takes a high-level design description, generates all switch configurations for the large-scale network, and publishes them to the relevant databases and repositories.

Matryoshka has been in production for six years, undergoing four major code revisions and growing to over 400K lines of code. It has been the foundation for designing, deploying, and continually evolving five generations of Meta's DCNs, powering close to 900 DCNs across 18 distinct types. Over this journey, Matryoshka has supported an over 10× increase in the number of DC types and an over 400× expansion in the number of DCNs, with the latest milestone of a 100K-GPU AI supercluster that underpins Meta's AI frontier and fuels its leadership in large-scale foundation models like LLaMa.

Today, Meta's DCN infrastructure is entirely built on Matryoshka. Its sustained success is rooted in a set of design

*Equal contribution.

†Work done at Max Planck Institute for Informatics.

principles that address practical challenges, proven effective throughout the system iterations and years of operation.

The first challenge arises from Meta's planet-scale DCN footprint—spanning 70+ regions, with multiple DC buildings and thousands of switches per building—forming one of the largest network infrastructures on Earth. The core design insight, which inspired the name Matryoshka, is grounded in two principles: (1) using Clos as a modular, reusable topology building block that can be replicated and adapted like nesting dolls to construct networks at massive scale, later extended to incorporate full-mesh topologies to meet the bandwidth density demands of AI clusters; and (2) defining a standardized configuration workflow that systematically generates switch configurations step-by-step, regardless of the network scale, mirroring the nesting procedure in Matryoshka.

The next challenge lies in the rapid growth and evolution of DCN infrastructure. This includes the greenfield construction of new DCNs, particularly over the past two years driven by the AI boom, where AI-dedicated networks have expanded from rack-scale to building-scale and even region-scale deployments. Alongside these new builds, existing DCNs are undergoing continual brownfield upgrades, including physical enhancements such as switch replacements and link speed increases, as well as routine operations like network migrations and switch platform transitions. These evolving architectures require new switch configurations and strong reliability guarantees, as upgrades must be performed live—without disrupting traffic or introducing misconfigurations.

Matryoshka meets these demands through deterministic and stateless intent compilation. As a stateless system, it delegates all state management to Meta's source-of-truth network database, FBNet [34], and its deterministic nature allows Matryoshka to always re-compile network design intents from scratch, independent of previous outputs. This design simplifies brownfield upgrades as greenfield deployments and ensure predictable network behaviors for straightforward correctness verification, troubleshooting, and error tracing.

The final challenge is managing heterogeneity across multiple dimensions, from diverse DCN types (e.g., general-purpose vs. AI clusters) to a broad range of vendor switches with varying software stacks, resulting in highly diverse configuration requirements. Matryoshka employs a generic network model to define high-level design intents that abstract away underlying differences. These intents are first translated into generic platform-agnostic configurations, and then automatically rendered into low-level, vendor-specific switch configurations. In practice, however, creating a fully generic model that captures all variations is infeasible, given the continual emergence of new DCN types and switches. Matryoshka thus combines abstract modeling with targeted codification of switch-specific requirements, striking a balance between system extensibility and adaptability.

The remainder of the paper is organized as follows. We begin by introducing the background of Meta's DCN infrastructure and switch configurations (§2), followed by an analysis of production data to quantify the challenges and define the design requirements for Matryoshka (§3). We then detail the above design principles (§4) and system architecture (§5) of Matryoshka. To demonstrate its practical impact, we present a case study of Meta's latest AI-specific DCN spanning 100K GPUs and show how Matryoshka enabled its design and evolution to meet Meta's AI ambitions (§6). We evaluate Matryoshka in production at scale (§7), share operational insights (§8), review related work (§9), and conclude the paper (§10).

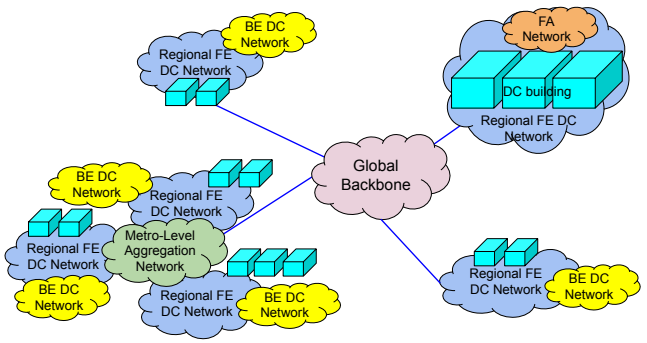


Figure 1: Meta network architecture.

To our knowledge, Matryoshka is the first production-grade DCN design realization system made accessible to academia. With Matryoshka, we aim to spark a new line of research that bridges industrial practice and academic innovation—encouraging practitioners to develop more efficient software tooling for configuration generation, and inspiring researchers to extend the scope of network management into the pre-deployment phase.

[This work does not raise any ethical issues.]

2 Background

This section provides an overview of Meta's hierarchical DCN architecture, key device types (§2.1), and the underlying network protocols and configurations (§2.2) that enable its large-scale, high-performance operations.

2.1 Meta DCN Infrastructure

DCN architecture. Meta's DCN infrastructure follows a hierarchical design that spans from individual server connections to global networks, as shown in Fig. 1. At the highest level, multiple DC regions are interconnected via backbone network, which consists of core and border components. The backbone border switches are located in each DC region and connect regions to one another through the backbone core. Sometimes for geographically proximate regions, a metro aggregation (MA) layer interconnects multiple regions, providing an intermediate layer between regional networks and the global backbone. The MA layers are part of the DCN infrastructure, rather than the backbone.

Inside a DC region, there are multiple fabric networks. Fabric networks can span across multiple DC buildings within a region. A fabric network is the one that hosts hundreds of thousands of servers serving various applications and providing different services. Those fabric networks are typically

interconnected via a fabric aggregation (FA) layer where inter-fabric but intra-region traffic is forwarded via the FA layer without going through backbone. The inter-region traffics go through a fabric → FA → backbone border → backbone core and then are delivered to their respective destinations.

DCN types. Meta's DCNs can be broadly categorized to traditional DCs and AI DCs. Within AI DC, there are two types of networks: the front-end (FE) that carries data and storage services and the back-end (BE) networks that carry AI training traffic [15]. Each BE includes four AI training clusters [13], each with ToR and aggregate layers. There are also direct links between the ToRs of FE and BE networks.

There are two types of FE networks: traditional FE (also called dcT), which follows same topology as traditional DCs, and power intensive FE (also called dcP), a newer generation design. The BE networks have a large number of varieties, as we need to customize designs for different space, power, performance, accelerator, and transport constraints. *Network types* are defined based on device roles, e.g., fabric, FA.

Topology types. Meta's DCNs mainly use Clos topologies, a multi-tier switching architecture offering non-blocking connectivity. The design has evolved from F4 [29] to F16 [5]. In the current dcT generation, each building's fabric has three layers: ToR, aggregate, and spine. The next-generation design (dcP FE) divides buildings into multiple smaller Mega-PODs, each with three layers. All Mega-PODs in a region connect through a regional spine layer of multiple switch groups.

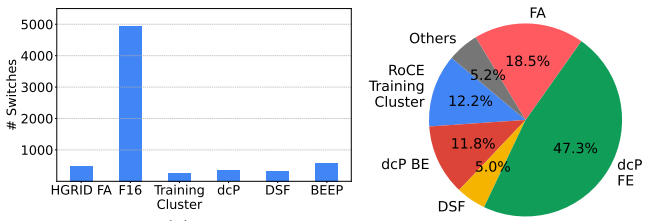
For AI training, Meta adopts specialized topologies tailored to the communication demands. Initially, Meta used a two-stage Clos topology organized into AI Zones. To support larger language models that require tens of thousands of GPUs, this architecture evolved into a multi-zone aggregation topology. As AI training continues to scale, Meta is also exploring rail-optimized and rail-only topologies [35].

Device types. Meta's DCN architecture uses a variety of specialized devices. Key devices include rack switches connecting to servers, fabric switches aggregating rack traffic within pods, and spine switches providing core connectivity between pods. For fabric aggregation, fabric aggregation downlink units and uplink units manage traffic between buildings and external networks. For AI workloads, Meta uses a hierarchical structure: aggregator training switches connect multiple AI Zones, cluster training switches serve as spine switches within zones, and rack training switches provide connectivity for GPUs within racks [14].

2.2 Switch Configurations

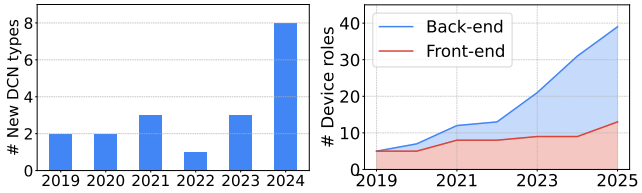
After outlining Meta's hierarchical DCN architectures, we proceed to discuss the configurations required on switches to realize a logical DCN design.

Switch Hardware and OS. Meta's DC is powered mostly by our in-house white-box switch FBOSS [9]. Designed as part of Meta's software-centric approach to networking, FBOSS runs on multiple hardware platforms. Configurations to FBOSS devices are done through Thrift interface to its management plane. It involves various hardware parameter settings. The configurations differ slightly across hardware platforms to best leverage the hardware capabilities. Besides FBOSS, a DCN also employs other vendor switches for specialized purposes, e.g., management and security. The functionalities of these devices are different from the main forwarding switches, and thus require different configurations.



(a) (b)

Figure 2: (a) Switch count by Network type. (b) Ratio of deployed DCs in 2025.



(a) (b)

Figure 3: (a) Number of new introduced DCN types. (b) Number of supported device roles.

Logical network. A switch consists of multiple physical ports mapping to physical interfaces, which can be bundled together to form aggregated interfaces (e.g., LAG) for increased bandwidth. Each interface, whether physical or aggregated, is assigned an IP address as a logical identifier. Additionally, a switch may have a loopback and separate management IPs for in-band and out-of-band control traffic. Interface properties include operating speed, maintenance state, circuit role, and uplink/downlink designation.

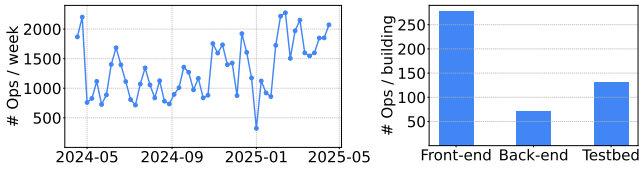
Routing. Meta's DCN uses multiple routing protocols. OpenR [1] handles intra-domain connectivity, while BGP is the main protocol for policy-based and inter-domain routing between autonomous systems. Supporting BGP requires defining autonomous systems and assigning autonomous system numbers, configuring BGP peering sessions, and generating routing policies to meet high-level intent [2, 28].

Other protocols. A switch configuration includes settings for various protocols beyond routing, such as: NTP for time synchronization, queue management for class of service and control traffic protection, and RDMA settings in AI clusters.

3 Challenges

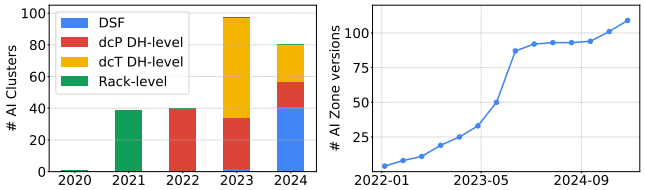
By examining measurement data from Meta's production deployments, we reveal evidence of three challenges: hyperscale, rapid evolution, and heterogeneity. The following analysis demonstrates how these factors shape the complexity of today's data center networks.

Challenge 1: Hyperscale DCN infrastructure. A DC contains a large number of switches, and different network types



(a) (b)

Figure 4: (a) Number of Matryoshka operations per week. (b) Number of Matryoshka operations per building.



(a) (b)

Figure 5: (a) Number of new deployed AI clusters. (b) Number of AI Zone versions.

have varied number of switches. Fig. 2a illustrates substantial variation in switch deployment across different network types. The sheer number indicates that network design has to rely on fast tools. At Meta, the back-end network already interconnects 100K GPUs, highlighting the enormous scale involved. Scaling to these sizes presents significant challenges for Matryoshka, which must ensure both scalability and efficiency at hyperscale. This wide disparity highlights the diverse architectural requirements. Fig. 2b shows the types of DCs and their distribution among the nearly 900 DCNs Matryoshka supports so far. Majority of DCs are categorized as dcP FE, accounting for the largest proportion at 47.3%, followed by FA (18.5%), and RoCE Training Clusters (12.2%).

Challenge 2: Rapid growth and evolution. Meta's DCN not only has large scale but also is evolving fast. Fig. 3b illustrates the growth in the number of supported device roles over time for both FE and BE networks. It illustrates an unprecedented growth trend due to the AI race. We further illustrates the growth based on the Matryoshka model changes, which is a reflection of network design intent changes.Fig. 4a shows the number of Matryoshka operations per week over the span of one year. The data reveals considerable week-to-week variation, with operation counts generally fluctuating between 800 and 2000. Several spikes are observed, particularly in early 2025, indicating periods of increased usage. Similarly, Fig. 4b breaks down the number of operations per building by network type. The majority of operations are associated with the front-end networks, followed by the testbed networks. This is because front-end networks frequently require migrations and dynamic changes to support evolving user-facing services, whereas back-end networks are more predetermined and stable, resulting in fewer operational calls. These frequent and unpredictable operational changes further challenge Matryoshka 's ability to ensure robust and efficient network management under constantly rapid growth and evolution.

Challenge 3: Heterogeneity across multiple dimensions. Fig. 3a presents the number of newly introduced data center types per year from 2019 to 2024. While most years remain relatively stable, 2024 sees a notable surge with the introduction of eight new data center types, indicating a significant acceleration in diversity. The growth in supported device roles shown in Fig. 3b further highlights the increasing heterogeneity of Meta's data center environments. This rising heterogeneity across both infrastructure and device roles creates a substantial challenge for network design.

The increase is particularly caused by the demand to support AI workload. Fig. 5a shows the number and types of AI clusters deployed annually from 2020 to 2024. The data reveals a shift toward more diverse and large-scale AI infrastructure. Fig. 5b displays the number of AI Zone versions over time, emphasizing the growing diversity and complexity of AI network designs. Each AI Zone version is defined as a unique network configuration. The sharp increase in versions beginning in mid-2023 reflects both the proliferation of AI workloads and the rising need for customized, rapidly evolving infrastructure solutions.

4 Design Principles

Over six years of production deployment, Matryoshka has guided the design, rollout, and continuous evolution of five generations of DCNs, powering a total of nearly 900 DCNs across 18 distinct types. Today, Meta's DCN infrastructure is 100% Matryoshka-based. As detailed in §6, Matryoshka underpins the regional AI supercluster comprising 100K GPUs and continues to drive the design and deployment of next-generation AI clusters at even greater scale.

The Matryoshka codebase has undergone 4 major revisions over the years. Its ability to support a substantial fleet of nearly 900 DCNs—marked by significant diversity and frequent architectural transformations, as revealed in §3—is rooted in critical design principles that have remained over the system iterations. In this section, we distill key lessons from the development and operation of Matryoshka, highlighting the core principles that have enabled its long-term success.

Modular topology construction. The design of modern DCNs fundamentally revolves around the topology. Production DCNs are built using Clos topologies with varying numbers of stages and scales, adding layers to interconnect more devices and adjusting connection densities to accommodate diverse bandwidth requirements. Matryoshka leverages this primary insight—reflected in its name—treating Clos as a modular topology building block. This design principle enables the construction of complex DCNs through the replication and adaptation of this basic structure, with necessary adjustments to scale and connectivity.

While Matryoshka has grown into a comprehensive software tooling system supporting a wide range of DCN designs, its origin lies in materializing Clos-based network design intents into concrete switch configurations. This core functionality enabled Matryoshka to later extend naturally to diverse DCN types and heterogeneous devices. As AI training clusters emerged with increasing demands for both high through-

put and low latency, Clos alone was proved insufficient—continuing scaling by adding hierarchical layers is infeasible facing the end-to-end latency constraint. Subsequently, we introduced the full-mesh topology as another fundamental building block. Yet, this initial design philosophy of using modular, composable topologies significantly facilitated Matryoshka's organic growth to meet new DCN design requirements.

Complete configuration workflow management. While topology is central to DCN design, realizing a complete DCN involves a comprehensive workflow beyond just topology specification. Prior work on DCN design focuses on topology only, discussing individual aspects of planning, deployment, and expansion [30,38,39]. However, implementing a DCN design extends far beyond physical tasks like switch installation and cabling. It requires generating a complete suite of switch configurations that precisely align with the physical deployment to ensure the network operates correctly and efficiently. Furthermore, the proliferation of heterogeneous devices and DCN types (§3) necessitates a reliable, single source of truth for consistent state management across all networks.

In Matryoshka, we for the first time model the complete workflow of switch configuration, from defining switches and their interconnectivity in the topology graph, to assigning IP addresses, allocating autonomous system numbers, and specifying BGP policies and routing configurations, as detailed in Fig. 6 and §5. We leverage FBNet, Meta's source-of-truth network database, to match logical network states and physical switch configurations. Matryoshka distinguishes itself from prior work by providing an end-to-end, automated realization of network design intents via switch configuration generation. This embodies the second meaning behind the Matryoshka name: to fully materialize DCN designs by progressively concretizing configurations—layer by layer—like nesting dolls.

Generic modeling of network intents. Matryoshka employs intent-based networking, where a software system acts as a compiler to translate high-level network design intents into concrete working configurations. In this context, Matryoshka serves as the compiler, and the design intents are expressed through an abstraction model implemented in form of Thrift `struct`. The use of Thrift, Meta's native Interface Definition Language and RPC framework, integrates Matryoshka into Meta's Thrift-based software stack, enabling rapid development with the rich set of software tooling.

The generality of the abstraction model is crucial. Ideally, it should be expressive enough to support new designs without requiring code changes to the compiler logic. However, in practice, achieving this level of generality is challenging, as some aspects of a DCN are more amenable to abstraction than others. For instance, network scale and topology are now well-modeled in the abstraction model which Matryoshka receives as inputs. On the other hand, we rely on IP allocation logic implemented in the compiler code section specific for the network type. In Matryoshka, we strive to capture as many network intents as possible through the generic input model, but we can make decision to implement a specific need in network type specific compiler section if the re-usability in future networks or possibility to capture in a future-proof generic model is less-likely. This approach strikes a practical balance between generality and specialization, keeping the Matryoshka codebase maintainable—even under major design heterogeneity, from general-purpose FE to AI-specific BE DCNs.

Deterministic and stateless intent compilation. We intentionally designed Matryoshka to be deterministic and stateless. Given the same network design inputs, Matryoshka always produces identical outputs, and each run re-compiles all network intents from scratch, without relying on prior outputs. This differs from a stateful approach in which the prior network state is provided as input to the configuration generator. These properties are critical for supporting rapid network evolution. Determinism allows Matryoshka to handle both greenfield and brownfield deployments alike, avoiding the complexity of maintaining separate design paths. Statelessness enables clean-slate upgrades in brownfield environments, reducing state dependencies and minimizing errors. Moreover, determinism is essential for operations—it ensures predictable network behavior and simplifies troubleshooting by making it easier to trace incorrect configurations back to faulty code or design intents.

Recompile-from-scratch avoids the complexity and maintenance burden of incremental approaches, which accumulate per-scenario code and ever-growing intent models as use cases expand. It may occasionally need targeted compiler updates for non-backward-compatible model changes, but the overhead is manageable and more sustainable than maintaining incremental pipelines. As a result, switch configurations are always generated from the latest design intent inputs. For updating network modeling data in FBNet, Matryoshka minimizes churn by diffing current and desired states to produce precise deltas of database entities, applying only the necessary changes.

Generic switch configuration interface. To address the heterogeneity of vendor devices, each with its own operating systems and configuration formats, Matryoshka introduces a platform-independent configuration model called Generic Switch Configuration (GSC). Matryoshka compiles platform-agnostic network designs, such as topology, IP addressing, and routing, into GSC, which is then translated into vendor-specific configurations by vendor-specific tools. All switches, regardless of vendor or OS, are represented in the unified GSC model struct defined using the Thrift language. This abstraction decouples logical network design from platform-specific details, simplifying the compilation pipeline and improving reliability. Validation is cleanly separated into two stages: verifying consistency between network intents and GSC, and between GSC and the final vendor-specific configurations.

5 Matryoshka Design

This section provides an overview of Matryoshka's architecture (§5.1) and end-to-end workflow, covering the core modeling layer(§5.2), the generation of network topologies and port mappings through a modular and reusable design process (§5.3), hierarchical IP address allocation for operational consistency (§5.4), routing configuration generation from declarative policies (§5.5). Operational details on database management (§A.1), data validation (§A.2), and rollout procedures (§A.3) are deferred to Appendix A.

5.1 Matryoshka Architecture Overview

Fig. 6 presents Matryoshka architecture, which also resembles the workflow of generating the detailed design of a DCN.

Intent Modeling: A *network specification*, codified in Thrift models, captures high-level design intent and rules for a DC network. It includes specifications for topology, IP addressing, routing, and switch port configurations to realize the underlying network topology. The specification is limited to versatile aspects of the network, excluding static or less changeable aspects like QoS, which are directly consumed as templates when generating switch configurations. The network specification covers all layers of the network, from hardware to logical connectivity and routing, and is fed into Matryoshka's software modules.

Network intents are specified and maintained programmatically, with revisions automatically sanity-checked, human-reviewed, and versioned with retrievable histories. A company-wide standardized tooling stack runs integration, regression, and validation on each change, blocking merges on failure unless formally approved. This workflow formalizes traceability, reliability, and reproducibility in intent management.

Intent Realization: Matryoshka builds the network from coarse-grained to finer grained, starting with individual switch nodes (*Switch Entity*), including hostname, hardware platform profile, and provisioning state. It then generates the network topology by constructing links between switches, including both logical links and physical port-to-port connectivity (*Topology Builder*). It then allocaties IP prefixes using a designed algorithm and generating routing configurations such as BGP peers and routes. The realized network designs are encapsulated in an internal data representation (IR) model, which enables parallel feature development and enhances troubleshooting. This process involves several stages, to ultimately produce network modeling data and individual switch configurations.

Modular Design Process. Matryoshka's design process is iterative, taking days to weeks, and involves managing data generated by each software module. The data is generated and stored in a relational DB called FBNet [34] using the *FBNet Generator and Uploader* component. A DC network is modeled in FBNet, primarily including switch, port, and IP entities. When there's a design change, Matryoshka computes the latest FBNet model data and constructs database transaction instructions to update the network state in FBNet. The instructions are based on the differences between the pre-update and post-update states of the network, and Matryoshka validates the new FBNet data against the network specification before sending it to the CRUD layer for fulfillment.

Switch Configuration Generation and Rollout: Matryoshka generates individual switch configurations based on IR data, first in a generic format and then converted to a native switch configuration format specific to the vendor and operating system. It ensures correctness through extensive validations, including data model validation before loading to the database and configuration validation after generation. Additionally, the configuration deployment process is continuously monitored to prevent failures and stale configurations.

5.2 Network Modeling Layer

At the core of Matryoshka's design lies the network modeling layer, which provides a powerful abstraction mechanism for expressing complex network designs in a concise, maintainable format. We define network product definition model in Thrift, also called Network Spec, which allows engineers to specify their design intent through a declarative approach comprehensively.

Switch Entity struct defines the naming and structure for network devices. It categorizes switches into different roles and establishes switch naming templates with specific tokens and value ranges.

Hardware Platform struct specifies the physical characteristics of network devices, including the chassis model, line-card model, and switch operating system. The chassis and line-card models are translated into entities in the FBNet database, from which we generate bills of materials (BOMs) for procurement and deployment. This section supports both Meta's in-house switches and third-party equipment, providing the information that Matryoshka uses to generate network modeling data and switch configurations. The configurations include only static settings such as interface and switch loopback IPs, link speed, BGP peers, and peer groups. They also include static settings for certain features. Real-time status, such as port up/down and device drain/undrain, is not included.

Provisioning State section specifies the deployment state for every switch in the DC network, representing where each device is in its lifecycle. For example, a switch is in `PROVISIONING` state during bootstrapping and quality assurance testing, and transitions to `IN_USE` state when ready for production traffic, allowing flexible support for various operational requirements and network upgrades without rebuilding the entire system.

Network Topology. Matryoshka models network topology using two complementary specifications: Abstract Topology and Portmap Template. Abstract Topology defines logical inter-layer connections, typically specifying the connectivity pattern (e.g. clos, mesh), number of wiring, bandwidth and the devices to include.

Port Assignment maps logical connections to physical in-

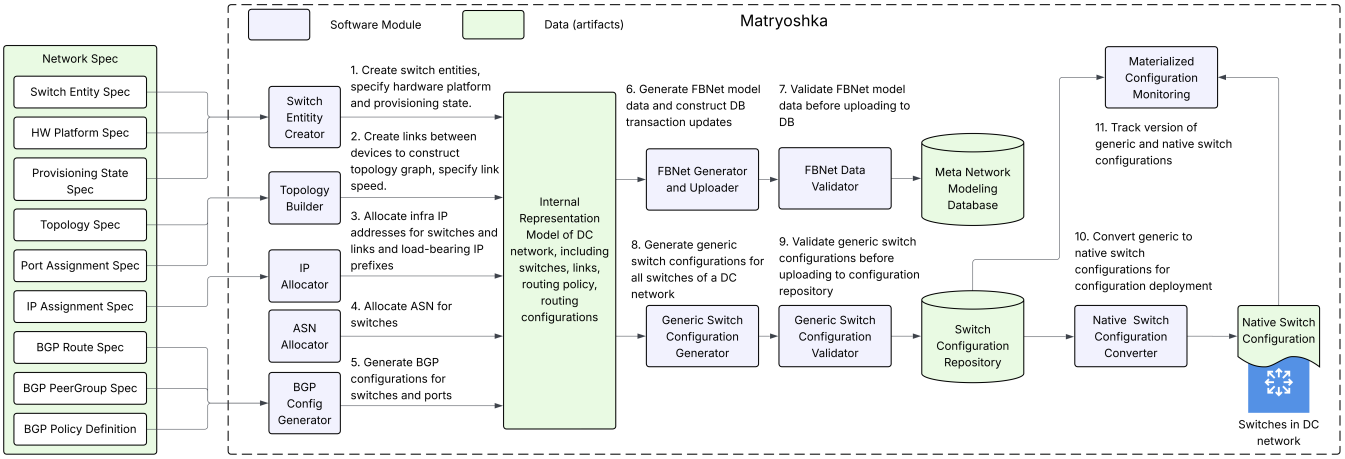


Figure 6: Matryoshka Architecture.

terfaces on network devices. The Portmap Template specification assigns physical ports to outgoing links based on predefined templates, transforming the abstract topology into a concrete implementation that can be deployed on physical hardware.

IP Assignment struct specifies the top-level IP subnets reserved for the particular DC network. While IP assignment at device/interface level is derived in Matryoshka codebase specific to the network type, this model constitutes the input for that particular logic.

Routing Configuration section describes the BGP routing setup via three key subsections: **BGP Route** (specifies the route templates for all member switches), **BGP PeerGroup** (specifies the BGP peergroup configurations for all member switches), and **BGP Policy Definition** (specifies the list of policies per session).

5.3 Topology Generation

The topology generation process generates the physical topology based on the high-level design intent. Here, design intent refers to the network engineer's desired outcome for network structure, connectivity patterns, and behavior, expressed as abstract specifications rather than detailed device-level configurations. This process handles both the logical structure of network connectivity and the mapping to physical ports, enabling network engineers to express complex topologies through simple, reusable patterns while accounting for hardware diversity and physical constraints.

Topology Categorization. Matryoshka distinguishes between two types of topologies: abstract and concrete. An abstract topology is a graph where vertices are real switch names but edge endpoints use generic interface names, which could be later mapped to a set of physical interfaces. A concrete topology maintains the same graph structure but uses actual physical interface names (e.g., "`EthX/Y/Z`") that will appear on physical switches. These two representations serve different purposes: abstract topologies help generate requirements for aggregated fiber paths and build of materials during early planning phase, while concrete topologies guide actual wiring during physical deployment.

Modular Design Approach. To address the challenge of rapidly evolving network designs, Matryoshka divides topology generation logic into product-independent and product-dependent components. The product-independent component provides a generic interface usable across all product designs, while the product-dependent component serves as a customization layer that transforms specific product requirements into a format compatible with the generic interface. This separation allows for flexibility in supporting new network types without requiring extensive code changes.

Topology composition. The key insight to support flexible topology is that any data center network can be represented by a mix of standard network topology blocks. The entire network can be considered as a set of topology groups. Topology Group is a group of standard network topology blocks with similar properties. For example, the layer of spine switches, fabric switches, and their interconnections form a bipartite graph. We define standard network topology Blocks as groups of switches interconnected in a standard pattern such as Ring, Mesh or Complete bi-partite graph. Thus, each device group is mapped to a basic connectivity pattern, called *graphlet*. In our production experiences, we found all our data center and PoP topologies can be covered by a combination of four graphlets: *Complete Bipartite, Ring, Mesh, and Circulant Bipartite*. Matryoshka supports a variety of topology by composing these graphlets flexibly. It does so via a top-down decomposition that partitions inter- and intra-layer connections into repeating blocks, each matching a predefined graphlet.

Two-Stage Generation Process. Matryoshka employs a two-stage process to convert high-level specifications into deployable network designs. First, it generates an abstract topology by extracting switch names from the specification to create vertices, converting topology specifications into switch clusters with defined connection patterns, and processing these clusters to create abstract topology edges. The abstract topology is then transformed into a concrete topology by applying portmap templates to map logical connections to physical switch ports. Edges are categorized based

on peer switch roles and real interface names are projected. This structured four-step procedure enables flexible handling of hardware diversity while maintaining consistent logical network structures.

Portmap Abstraction. A portmap specifies input and output ports for each device, connecting outputs from one device to inputs of another. In our implementation, we initially create network-specific templates, where switches were organized into device groups and templates determine port-to-port connections. However, this approach led to template proliferation and management challenges across hundreds of production networks. We enhanced reusability by replacing specific switch names with generic device identifiers, enabling template reuse across different network instances and significantly reducing the template count.

5.4 IP Addressing

Matryoshka's approach to IP Addressing follows hierarchical principles while prioritizing consistency and stability through network changes. It allocates two IP categories: (1) *Infra prefixes*, used for routing control traffic, include device loopback prefixes and port-level interconnect prefixes along with their aggregates; and (2) *Load-bearing prefixes*, used for routing data traffic, comprise dynamic cluster, VIP, and IP-per-task subcategories. A consistent subdivision approach is applied to each subcategory, dividing the top-level subnet into chunks until reaching per-device or per-port allocation.

5.5 Routing Configuration

Configuration Categories. BGP routing configuration is divided into two parts: *policy configuration* (containing definitions of communities, preferences, paths, filters, and policies) and *routing configuration* (settings attached to interfaces and prefixes). A common routing policy is applied to all member switches. To maximize flexibility, Matryoshka accepts routing policy as an external input.

Routing Configuration has two key components: *BGP peer configurations* and *BGP route configurations*. The former are pre-defined peergroup templates based on local-to-peer device role pairs (e.g., Spine-to-Aggregate). The later programmatically allocates prefixes using prefix category types in route templates. Each route template includes a network domain config for specific switches, providing fine-grained control over route distribution.

Operational details for data management, validation, and rollout are summarized in Appendix A.

6 AI Case Study: 100K-GPU Supercluster

Over the course of 2024, Meta deployed one of the world's largest AI superclusters, which boasts over 100K GPUs. This infrastructure has powered Meta's latest LLaMA model training and underpins its pursuit of artificial general intelligence. In this section, we use this AI supercluster as a case study to demonstrate how Matryoshka enables rapid iteration and deployment of DCN designs in the AI era.

Architecture overview. As noted in §2.1, we built back-end (BE) networks specifically for distributed training, enabling them to evolve, operate, and scale independently from the general-purpose front-end (FE) DCNs. To support large language models (LLMs), we have since expanded BE networks to full DC and even regional scale.

As illustrated in Fig. 7a, training clusters leverage both FE and BE networks. Each training rack connects to the FE network for tasks like data ingestion, checkpointing, and logging via the hierarchical switch layers detailed in §2.1. Simultaneously, it connects to the BE network through a two-stage Clos topology (Fig. 7b), forming the so-called AI Zones.

Within each AI Zone, rack training switches provide scale-up connectivity to GPUs within racks and uplink to cluster training switches, which deliver scale-out connectivity across racks. This design enables non-blocking GPU interconnect within an AI Zone for maximum performance. AI Zones arocss DC building are interconnected via aggregator training switches through a full-mesh topology, but with intentionally oversubscribed bandwidth to maximize port usage supporting high GPU connectivity density.

This BE network provides high-capacity, low-latency, and lossless transport between any two GPUs within the supercluster through RDMA-enabled NICs using the RoCEv2 protocol.

FE brownfield migration. To accelerate development, this AI supercluster was repurposed from an existing FE network. It consists of three buildings, originally deployed as dcT Fabric networks and interconnected via an FA. Each building previously hosted four RoCE clusters, supporting storage and a variety of services not exclusively dedicated to AI training.

To enable dense GPU placement and optimize training locality, the original four RoCE clusters were expanded into eight AI Zones. Once the buildings were fully populated with GPU racks, storage and services were relocated to a separate building. In contrast to traditional FE network applications such as Facebook, Instagram, and WhatsApp, large-scale training jobs demand significantly higher bandwidth to avoid performance bottlenecks. Since the storage warehouse, which supplies input data for training, now resides in a different building, cross-building traffic has increased substantially, necessitating a bandwidth upgrade in the FA.

This upgrade was seamlessly executed using Matryoshka, leveraging its native support for brownfield evolution through simple network intent modifications (§5.2). For instance, upgrading FA capacity required only updates to the network specification: expanding the number of device groups, specifying backbone connectivity, and adjusting port mappings to integrate the new FA devices. Similarly, relocating storage and compute nodes involved redefining the network specification for the new building. Matryoshka automatically translated these changes into switch configurations, mirroring the physical deployment.

It was essential to maintain continuous operation of the FE network throughout the migrations. This was ensured by Matryoshka's deterministic and stateless design (§4), which

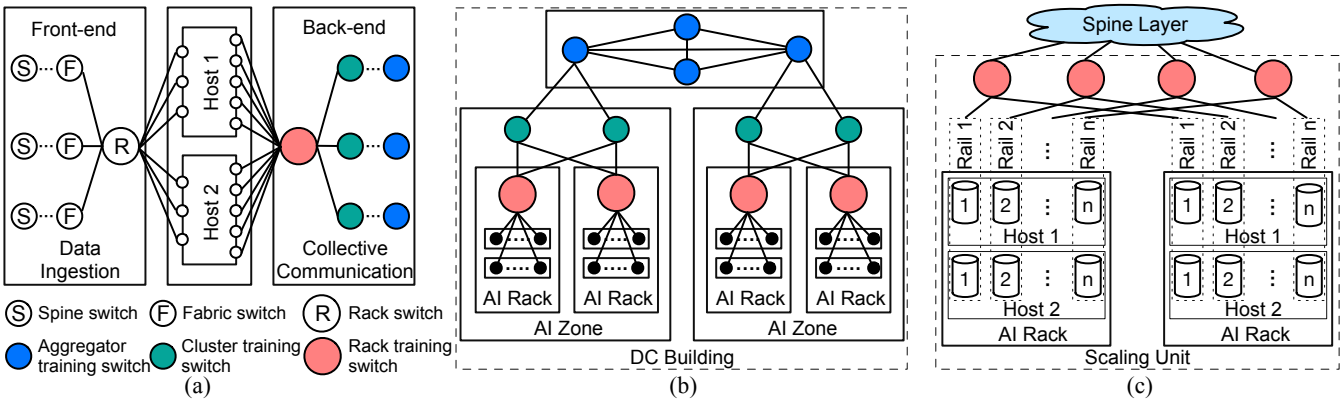


Figure 7: (a) Front-end and back-end networks. (b) The 100K-GPU supercluster. (c) Rail-based topology.

compiles network intents from scratch, treating brownfield upgrades as greenfield deployments. During this process, Matryoshka's compile-time data validation guaranteed correctness by verifying that configurations for unaffected switches remained unchanged and by cross-referencing state information in FBNet to validate the consistency of all modifications.

BE greenfield construction. Two additional AI cluster buildings were constructed alongside the three remodeled ones, resulting in a total of six buildings—five dedicated to AI and one to storage. This expansion, like the earlier brownfield migration, was driven through network specifications.

Although the BE network is specialized for training, its greenfield construction shares key similarities with traditional FE DCNs, allowing Matryoshka to seamlessly adapt existing specification templates. Clos and mesh topologies—both natively supported by Matryoshka (§4)—form the foundation of the BE design. Besides, rack switches, cluster training switches, and aggregator training switches mirror the roles of rack switches, fabric switches, and spine switches in FE networks (§2.1). Matryoshka's native greenfield support and modular architecture streamlined the AI cluster buildout, making the process efficient and straightforward.

Modeling new topology and wiring configuration. Despite structural similarities, the BE network must model the entire regional supercluster as a unified entity to support large-scale cross-DC training jobs, unlike FE DCNs, which model individual buildings and the FA separately. Depending on wiring constraints, some AI Zones deploy rack training switches directly within each rack, connected to GPUs via copper direct-attach cables. Others use an End-of-Row design, consolidating rack training switches into a separate rack and connecting them to GPUs via optical transceivers and single-mode fibers. These new topologies and wiring schemes must be accurately captured in the network model.

The regional topology was straightforwardly modeled by stacking modular components, combining pre-defined Clos and full-mesh topologies into a unified network specification. The End-of-Row wiring required additional details on optical transceivers and fibers, necessitating new specification structures and corresponding FBNet states. Its port mapping process was well-defined, backed by our powerful topology library. Thanks to Matryoshka's modular design, these changes are self-contained and isolated from other parts of the DCN, enabling safe and incremental updates.

Handling greater heterogeneity. AI clusters exhibit unprecedented heterogeneity, spanning 109 cluster versions (Fig. 5b) and 26 device roles (Fig. 3b). This reflects the rapid evolution of BE DCN designs alongside a diverse hardware landscape—ranging from successive generations of NVIDIA GPUs to Meta's MTIA accelerators, from external vendor fabrics to in-house rack switches and RDMA fabrics, and from vendor-specific switch OSes to Meta's native FBOSS. AI workload diversity, such as GenAI vs. non-GenAI, further drives varying topology designs and inter-GPU bandwidth requirements.

As explained in §4, the generic switch configuration (GSC) effectively masks underlying heterogeneity, enabling natural support across diverse hardware types and switch operating systems. Switch configurations are first generated in a uniform, logical GSC format and then translated into platform-specific formats, ensuring that each device—regardless of vendor or OS—receives the correct, customized configuration. This design makes Matryoshka inherently resilient to the current and future heterogeneity of AI clusters, providing long-term extensibility as infrastructure continues to evolve.

Support for rapid topology evolution. The architecture of our BE networks has gone through significantly transformations over time. Early GPU clusters were built using a basic star topology, where a small number of AI racks connected to a central Ethernet switch. This design posed limitations in both scalability and redundancy. To address these challenges, we rapidly adopted the AI Zone fabric-based architecture to achieve greater scalability and improved availability.

As for now, the supercluster relies on one-to-one mapping between AI Racks and rack training switches (Fig. 7b), in which case all GPUs in a given rack uplink to the same rack training switch. Other BE deployments at Meta have explored alternative technologies and architectures that offer distinct performance advantages. We introduce two notable examples as follows.

Table 1: Matryoshka output correctness in recent one year.

Success count	Failure count	Success rate
12100	589	95.4%

Table 2: Matryoshka behavior correctness since 10/23/2024.

Success count	Failure count	Non-system failures	Success rate
568	46	25	96.4%

The first is the Rail-based Topology, as illustrated in Fig. 7c, where GPUs in the same rank position from multiple hosts uplink to the same leaf switch. The grouping of AI Racks and their corresponding leaf switches that participate in this connectivity scheme is referred to as the scaling unit (SU). Rail-based topologies are optimized for workloads that rely on communication among GPUs within the same rail, such as the all-reduce collective. By first-hop switching traffic within each rail at the leaf switch, this topology can significantly reduce latency. A higher-level spine tier interconnects rails within an SU and links multiple SUs together.

Another example is the Distributed Scheduled Fabric (DSF), which combines dedicated leaf and fabric switches to form a large, distributed switching system. Each leaf switch has both Ethernet ports for downlink connectivity to the NICs and fabric ports for uplink connectivity to fabric switches. The leaf switch breaks down packets arriving from the GPUs into cells, and sprays the cell over the fabric uplinks, achieving even load balancing. A custom header carrying metadata is added to each cell to enable forwarding the cell by the fabric switch and reassembling the packet at the receiver leaf end. Virtual Output Queues are set up on all leaf nodes to efficiently buffer traffic and schedule transmission when credit is available to send.

DSF networks introduce new requirements for configuration generation. For example, peering of the routing layer is performed between leaf nodes, unlike other networks through the spine. IP allocation should be disabled on DSF fabric nodes and for its connections. Inter-connection of DSF clusters into building-scale superclusters is performed in different topologies. Matryoshka handles this heterogeneity gracefully, since each of those components (routing, IP, topology etc.) are isolated from each other. Users only need to touch on the layer that is different and utilize the same wiring, device definition, Clos topology building, and IP allocation (for devices applicable) methods.

Matryoshka's building block of Clos topology (§4) naturally accommodates variations. It supports interconnecting leaf nodes for DC-scale topologies for DSF, as well as rail-based wiring, which interconnects GPUs at the same rank across hosts, departing from traditional intra-rack grouping.

7 Performance Evaluation

In this section, we evaluate the performance of Matryoshka in large-scale, production environments. Our evaluation focuses on three key aspects: correctness (§7.1), efficiency (§7.2), and scalability (§7.3).

Table 3: Performance gain of Matryoshka FBNet updating algorithm for FA networks.

FA network operation	Transaction time of Version 1	Transaction time of Version 2	Reduction of transaction time
Add 88 switches + hundreds of links	200 seconds	60 seconds	70.0%
Add 288 links	37 seconds	3 seconds	91.9%

Table 4: Performance gain of Matryoshka FBNet updating algorithm for fabric networks.

Fabric network operation	Transaction time of Version 1	Transaction time of Version 2	Reduction of transaction time
FA expansion	34.60 minutes	46.00 seconds	97.80%
Grid move	34.84 minutes	13.19 seconds	99.37%

7.1 Correctness

Matryoshka output correctness. Output correctness refers to whether the Generic Switch Configuration (GSC) generated by Matryoshka conform to expected standards and requirements. The Configuration Mover Service is a centralized system used in DCNs to verify these GSCs before deployment to production network. The Configuration Mover Service verification failure count reflects the number of times Matryoshka-generated GSCs have violated predefined rules, such as incorrect port speeds, routing policy errors, and other configuration issues. As shown in Table 1, the success rate for GSC verification has been 95.4% over the past year. In the case of configuration error, Configuration Mover Service will detect it, prevent the GSCs from being pushed to production network devices, and create signals to notify the network engineers.

Matryoshka behavior correctness. Behavior correctness in the Matryoshka system refers to its ability to function as intended. To ensure this, end-to-end tests are conducted daily to verify that Matryoshka produces the expected network configuration outputs based on predefined network intent inputs. These tests assess detailed configuration information, including but not limited to network topology, address, port allocation, routing policies, and hardware models. As shown in Table 2, the success rate of Matryoshka has been 96.4% since October 23, 2024. This indicates that the Matryoshka system is highly reliable and robust, even in the face of frequent code changes. In the event of unexpected behavior, the end-to-end tests will capture it and prevent the system changes from being deployed in the Matryoshka production release.

7.2 Efficiency

FBNet is Meta's MySQL-backed database system that models the entire network infrastructure. To assess the efficiency improvements brought by Matryoshka's generic FBNet updating algorithm, we compare the performance of its two major versions: Version 1, which deletes and recreates the entire model, and Version 2, which applies incremental updates by computing differences between pre- and post-update states.

Table 3 and Table 4 show the reduction in transaction time achieved by Version 2 for two types of networks: FA and fabric. In FA networks, which typically contain over 300 switches and more than 2000 links, Version 2 significantly

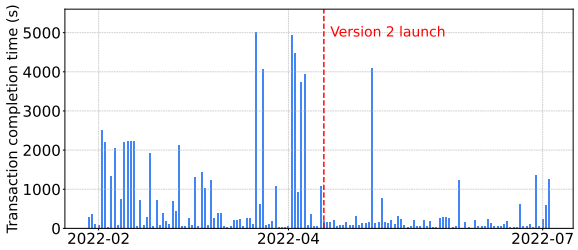


Figure 8: Matryoshka FBNet transaction completion time.

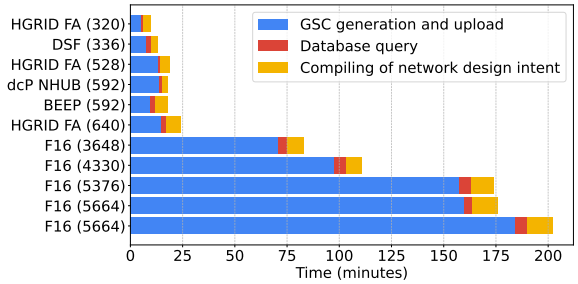


Figure 9: Processing time breakdown across network types (numbers in parentheses indicate network scale).

reduces update latency. For instance, adding 88 switches and hundreds of links takes 200 seconds with Version 1 and only 60 seconds with Version 2, resulting in a 70.0% time reduction. A more fine-grained operation, adding 288 links, showed a 91.9% reduction, from 37 seconds to just 3 seconds.

For fabric networks (Table 4), which feature more complex interconnectivity and abstract topology management, the improvements are even more striking. For FA expansion operation, transaction time is reduced from 34.60 minutes to 46.00 seconds, yielding a 97.80% reduction. Similarly, a grid move operation drops from 34.84 minutes to 13.19 seconds, corresponding to a 99.37% reduction.

Fig. 8 illustrates the impact of deploying Version 2. It shows the transaction completion times over a six-month period in 2022. The red dashed line marks the deployment date of Version 2 on April 27, 2022. Before this date, transaction times frequently spike, often exceeding 2000 seconds. After Version 2 was launched, the spikes largely disappear and transaction times stabilize at significantly lower levels, confirming the practical performance gain brought by the incremental update algorithm. Notably, some spikes remain after Version 2's launch because incremental updates are only used for brownfield, while greenfield still require full updates, resulting in similar transaction times for both versions in those cases. This explains why a few spikes persist in the figure, though they are much less frequent than before.

7.3 Scalability

To capture the evolution in both scale and diversity of deployed DC networks, Fig. 10 presents both the number of DCN types and the total counts deployed from 2019 to 2025. The left y-axis quantifies the diversity of network types, while the right y-axis tracks the aggregate number of individual DC networks. The data indicate an upward trend in both metrics, with the number of DC network types increasing from 2 in

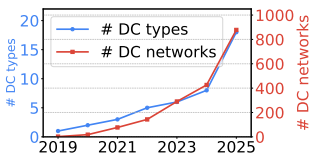


Figure 10: Evolution of DCN types and total deployments.

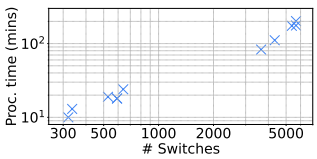


Figure 11: Processing time as a function of network size.

2020 to 18 in 2025, and the total number of DC networks expanding from 18 to nearly 900 over the same period. Notably, both metrics exhibit accelerated growth beginning in 2023, reflecting a period of rapid architectural diversification and large-scale deployment.

To assess the scalability of Matryoshka, we measure the total run time required to generate configuration files for all devices within a network and upload these configurations to the repository, using the same command type across experiments. For this evaluation, we focus on network types where the end-to-end processing time exceeds 10 minutes, encompassing network scales ranging from 320 to 5664 switches per network. We further decompose the total processing time into three major components: (1) GSC generation and upload, (2) database query operations, and (3) compiling of network design intent. This breakdown is visualized in Fig. 9. As the network size grows, all three components exhibit increased processing times, but they scale at different rates due to the distinct nature of their workloads.

The GSC generation and upload stage, which operates on a per-switch basis, becomes the dominant contributor to total processing time for larger networks. Each switch requires a roughly constant amount of time to generate its generic switch configuration, resulting in a processing time that grows strictly proportionally with the number of switches in the network. In contrast, both the database query and the compiling of network design intent operate primarily on a per-network basis. Although they still involve generating attributes for individual switches, the computational load for these components is significantly lower compared to GSC generation and upload. Consequently, while their processing times do increase as the network size grows, their growth is much more moderate.

Fig. 11 demonstrates the scalability of Matryoshka by presenting the processing time as a function of network size (measured by the number of switches). The logarithmic axes capture the range of both variables. Notably, the data points exhibit an approximately linear trend, indicating that the processing time increases proportionally with network scale. The trend is linear because processing time scales with both the number and size of GSCs. The GSC count grows with device count, and around 90% of Matryoshka 's CPU time is spent serializing/deserializing GSCs. This near-linear relationship highlights the inherent scalability of Matryoshka, confirming that the system can efficiently handle increasingly large data center networks without disproportionate increases in processing overhead.

8 Operational Experiences

We share six years of experience operating Matryoshka in production networks, with valuable lessons learned from real-world incidents.

Large blast radius. Matryoshka has a wide blast radius, operating across multiple DCNs and device types, where a single code error can misconfigure thousands of devices. For instance, a bug in Matryoshka during a brownfield migration in 2020 accidentally deleted all IPv6 prefixes for inter-fabric circuits, risking misconfiguration of all fabric switches in the affected region. This incident prompted a major investment in strengthening Matryoshka's system validation, significantly improving its reliability, as shown in our evaluation (§7.1).

Interoperability with other tools. Matryoshka is the main system for generating data to FBNet for DC design, but other tools also modify switch states for maintenance. During a lengthy DB update by Matryoshka, a maintenance tool read an intermediate inconsistent state, mistook it for device failure, and triggered false alarms that shut down devices. This incident motivated a shift from delete-then-recreate to incremental updates with deltas, reducing database transaction time by up to 99.37%, as detailed in §7.2. As a more fundamental solution, we have set standards for atomicity of DB transactions via both Matryoshka and FBNet service infrastructures.

Shared code path across different DC types. Using a single tool for all DC types enables reuse and reduces redundant development. However, changes for one DC type can introduce bugs in others. For example, adding portmap support for AI BE clusters broke the logic for traditional DCs, causing failures during migration. This incident reminded us to be mindful of different natures of DCN types, creating dedicated test cases to thoroughly validate all design aspects. The problem has been addressed by setting end-to-end non-regression standards for all DCN types we add support for.

Abstract invariant for validation. In one case, adding a BGP policy to block certain prefixes unintentionally caused prefix mis-filtering in other DCs because of shared code paths. This highlighted the need for validating Matryoshka 's output, such as checking for critical prefixes. However, since Matryoshka generates these prefixes, it is difficult to define exact rules in advance. To address this, rules can be defined more abstractly, using naming conventions or prefix mask length heuristics, which breaks the interdependency between Matryoshka and validation.

Evolving to meet new challenges. Matryoshka faces the challenge of rapidly introducing new DC network products through the New Product Introduction process, especially for AI training clusters with unique needs. The introduction timeline has decreased from one year to four weeks. To improve development efficiency, Matryoshka has taken three approaches: improving network specifications for adaptability, enforcing common design standards to avoid exceptions, and developing shared algorithms like IP address assignment across different DC types.

9 Related Work

DCN design. The design of DCNs has been extensively studied in the past decade, with proposals of Clos [4, 16] and its variants [24, 39], as well as novel topologies [3, 17, 18, 22, 27, 32]. Matryoshka takes Clos and full-mesh as basic topology building blocks, covering the needs of both general services and AI training, and it can be extended to incorporate additional topologies as new requirements emerge.

Jupiter describes Google's five generations of DCNs and their decentralized routing and management protocols [31]. Condor proposes a declarative tool for selecting optimal topologies [30], while Malt presents a system for representing and storing network topologies at Google [25], and Robotron outlines Meta's network management software suite [34]. Other efforts have explored minimum rewiring strategies for live DCN expansion [39] and full life cycle management from deployment to scaling [38]. These works mainly focus on topology, either logical topology design and presentation [25, 30, 34] or physical deployment and expansion [31, 38, 39], but largely overlook the critical step of translating logical designs into switch configurations. Matryoshka is orthogonal to these efforts and fills this important gap.

Network management. Switch configurations have been extensively used for network verification [6–8, 10, 12, 19–21, 23, 33, 36, 37]. Matryoshka is orthogonal to this line of work, focusing on configuration generation rather than post-generation analysis. Matryoshka's compile-time validator serves a similar purpose, it provides more targeted and efficient validation tailored to production needs.

Matryoshka does not propose a new network design; it is orthogonal to prior work on routing, management protocols, switch configurations, rewiring, and lifecycle management. To our knowledge, it is the first academic study on DCN design realization.

10 Conclusion

We are proud to present Matryoshka, Meta's DCN design automation system. Over the past six years, Matryoshka has been carefully crafted to meet the challenges of operating massive-scale DCN infrastructures and addressing rapid growth, continuous evolution, and rising heterogeneity, particularly in the era of AI. By compiling switch configurations systematically from high-level DCN design intents, Matryoshka has successfully supported a broad spectrum of network types and scaled with the explosive expansion of AI clusters, including our latest 100K-GPU deployment. We believe our experience with Matryoshka offers valuable insights to the broader network design community and serves as a foundation for future innovations. We hope this work will inspire continued research and development toward the next generation of intelligent, scalable network design systems.

References

[1] Open/R: Open routing for modern networks. https://engineering.fb.com/2017/11/15/connectivity/open-r-open-routing-for-modern-networks/.

[2] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. Running BGP in data centers at scale. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 65–81. USENIX Association, April 2021.

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[5] Alexey Andreyev, Xu Wang, and Alex Eckert. Reinventing facebook's data center network. https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/.

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[8] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 122–135, 2023.

[9] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 342–356. ACM, 2018.

[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, April 2018. USENIX Association.

[11] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. Gpts are gpts: An early look at the labor market impact potential of large language models, 2023.

[12] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM Sigplan Notices*, 46(9):279–291, 2011.

[13] Adi Gangidi and James Hongyi Zeng. Roce networks for distributed ai training at scale. https://engineering.fb.com/2024/08/05/data-center-engineering/roce-network-distributed-ai-training-at-scale/.

[14] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 57–70, 2024.

[15] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[16] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[17] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 63–74, 2009.

[18] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 75–86, 2008.

[19] Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y Richard Yang. Flash: fast, consistent data plane verification for large-scale network settings. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 314–335, 2022.

[20] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, 2017.

[21] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, et al. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 200–213. 2019.

[22] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 281–294, 2017.

[23] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 9, USA, 2012. USENIX Association.

[24] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A {Fault-Tolerant} engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, 2013.

[25] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 403–418, 2020.

[26] OpenAI. Gpt-4 technical report, 2023.

[27] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 691–706, 2024.

[28] Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, Walid Taha, Minlan Yu, and Jelena Mirkovic. Practical intent-driven routing configuration synthesis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 629–644, 2023.

[29] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[30] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. Condor: Better topologies through declarative design. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2015.

[31] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Sigcomm '15*, 2015.

[32] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, 2012.

[33] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 572–585, 2017.

[34] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439. ACM, 2016.

[35] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. Rail-only: A low-cost high-performance network for training llms with trillion parameters. In *2024 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 1–10. IEEE, 2024.

[36] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2015.

[37] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and

Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 87–99, 2014.

[38] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding lifecycle management complexity of datacenter topologies. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 235–254, 2019.

[39] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 221–234, 2019.

Appendix

A Operational Pipeline: Database, Validation, and Rollout

A.1 Database Management

Matryoshka transforms high-level network specifications into database records and device configurations, enabling efficient updates.

Network Modeling Data. Matryoshka generates data to store in a MySQL DB called FBNet [34]. It store network entities across different tables with relationships expressed through foreign key associations. This database primarily models two categories: logical topology (switches, platforms, linecards, interfaces, and links) and prefix hierarch. This representation supports downstream systems that require specific network attributes, such as monitoring and configuration systems. A local copy of all DB row entities is maintained in-process, allowing for generation of new network models, retrieval of existing models from the database, and assessment of database changes to be executed.

Generic DB Updating Algorithm. When networks evolve, Matryoshka uses a 6-step algorithm to efficiently update the database: 1) Compare: Compute differences between source and target models; 2) Unlink: Update foreign key columns to None if necessary; 3) Delete: Remove rows no longer needed; 4) Update-static: Update non-foreign-key columns; 5) Create: Add new rows; 6) Link: Reestablish foreign key relationships. This approach generates transactions proportional to the change scope, enabling rapid updates for large networks.

FBNet Robustness. FBNet strengthens robustness with fine-grained access control, application-level validation, race condition mitigation, automated remediation, and comprehensive error handling with retries and graceful degradation. Availability comes from a globally distributed, multi-region primary–replica architecture with global load balancing and multi-tier services, providing read-after-write consistency and historical queries. Scalability follows from shorter queue timeouts, faster responses, in-memory caching, query optimizations, and horizontal scaling via read replicas. Writes are strongly consistent with transactions, while replica reads are eventually consistent. Rigorous validation and auditing maintain data integrity while balancing performance, correctness, and resilience at scale.

A.2 Data Validation

Matryoshka employs a two-step validation process to prevent erroneous configurations from reaching production environments: 1) *FBNet Data Validation*, which verifies network topology statistics (switch count, link count, IP prefix count) against the network spec; 2) *Switch Configuration Validation* which checks completeness and consistency of switch configurations.

One example of validation is routing configuration. It cross-checks external routing policy definitions against generated peer information. Another example is routing emulation system, which validates network configuration against predefined criteria (e.g., no routing loops or black holes). Both verification processes are modular, allowing partner teams to extend them with their own validations.

A.3 Rollout Procedure

The switch configuration rollout process for a DC network consists of two steps. First, the latest network intent is compiled to generate new switch configurations, which are then staged in a distributed repository. The GSC configurations for the entire network are validated, and upon successful validation, they are moved to a production-ready directory. This process is triggered on-demand whenever there is a change in the network design intent.

In the second step, a configuration deployment event is kicked off to activate selected versions of switch configurations on the member switches of the network. The GSC-to-native switch configuration converter picks up the configurations and converts them to native switch configurations, which are then stored on the switches. Network operators arrange a deployment event to activate the new configurations. On detecting any adverse impact from newly deployed switch configs, fleet-wide rollout is halted. Offending configs are removed from the production repository or replaced with fixed versions; on devices, buggy configs are rolled back to a previous validated version or overridden with a hotfix.