Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

GaussCast: Dependency-Aware Shared Block Retrieval for Multi-User Layered 3D Gaussian Splatting

Yulong Zhang
The Hong Kong University of Science and Technology, Guangzhou
Ruonan Chai
The Hong Kong University of Science and Technology, Guangzhou
Jiadong Yu
The Hong Kong University of Science and Technology, Guangzhou
Zili Meng
The Hong Kong University of Science and Technology
Dirk Kutscher
dku@hkust-gz.edu.cn
The Hong Kong University of Science and Technology, Guangzhou

Research Article

Keywords: 3D Gaussian Splatting, multi-user delivery, progressive streaming, Information-Centric Networking

Posted Date: September 2nd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10239330/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

GaussCast: Dependency-Aware Shared Block Retrieval for Multi-User Layered 3D Gaussian Splatting

Yulong Zhang[1], Ruonan Chai[1], Jiadong Yu[1], Zili Meng[2], Dirk Kutscher[1*]

[1]The Hong Kong University of Science and Technology, Guangzhou, Guangzhou, China.
[2]The Hong Kong University of Science and Technology, Hong Kong, China.

*Corresponding author(s). E-mail(s): dku@hkust-gz.edu.cn;
Contributing authors: yzhang893@connect.hkust-gz.edu.cn; rchai327@connect.hkust-gz.edu.cn; jiadongyu@hkust-gz.edu.cn; zilim@ust.hk;

Abstract

Layered 3D Gaussian Splatting (3DGS) enables progressive novel-view synthesis, but large scenes such as city models remain gigabytes in size even after state-of-the-art compression, making full pre-download impractical under bandwidth limits or sub-second startup targets. Progressive delivery transmits content incrementally according to viewing demand, but existing systems plan it for each user independently. When several users explore the same scene concurrently, as in virtual tours or XR classrooms, their viewpoints differ yet depend on the same lower-layer blocks, so per-user delivery fetches the same prerequisites from the origin once per user, duplicating shared upstream traffic. Edge caches and request aggregation remove part of this duplication, but they act only after requests exist: they neither plan which blocks a group should fetch jointly nor respect rendering dependencies, so a refinement block can arrive before its required lower layers and remain unusable. We present GaussCast, a dependency-aware delivery framework that structures layered 3DGS scenes as versioned cell–layer–chunk retrieval blocks with explicit dependency metadata. An edge proxy aggregates short-horizon user demand, fetches a shared base of widely needed prerequisite blocks once upstream, and spends the remaining budget on per-user supplements; admission enforces dependency closure

under per-window byte budgets and user rendering deadlines. We implement GaussCast over Information-Centric Networking and describe mappings to HTTP/CDN and relay deployments. In trace-driven evaluation with multi-user sessions derived from real 6DoF navigation traces, GaussCast reduces shared edge-to-origin upstream traffic by 26% and time-to-first-frame by 27%, and improves viewport quality and fairness over per-user baselines, with gains growing as cross-user overlap increases. Implementation and evaluation artifacts are publicly available at https://github.com/ulen2000/gausscast-sim.

Keywords: 3D Gaussian Splatting, multi-user delivery, progressive streaming, Information-Centric Networking

1 Introduction

3D Gaussian Splatting (3DGS) represents scenes as explicit Gaussian primitives, enabling high-quality novel-view synthesis at interactive rates [1]. Compression techniques—including anchors, spatial hierarchies, pruning, quantization, and entropy coding [2–11]—have substantially reduced scene sizes. For instance, HAC++ [10] compresses certain scenes to under 10 MB, making full download feasible for many single-room or object-centric scenarios. Compression alone, however, does not eliminate the delivery challenge for all deployment contexts. Large-scale environments—city models, multi-room building interiors, and detailed outdoor scenes—remain in the gigabyte range even after state-of-the-art compression [4, 5, 12]. For such scenes, downloading the full compressed model before interaction is impractical under bandwidth constraints or sub-second startup targets; for example, mobile Extended Reality (XR) over a 10–20 Mbps link cannot pre-fetch a multi-hundred-megabyte scene while targeting sub-second startup. Progressive delivery addresses this by transmitting scene content incrementally according to predicted viewing demand. Recent layered 3DGS systems show that progressive transmission improves the viewing experience when bandwidth, latency, or scene scale preclude full pre-download [12–14].

Existing progressive 3DGS systems optimize single-user streaming. We address a complementary problem that arises when multiple users concurrently access the same scene—common in virtual tours, collaborative design reviews, XR classrooms, and digital-twin exploration. Although users render different viewpoints, they frequently depend on overlapping lower-layer scene data. Per-user delivery sends these shared prerequisites separately for each user over the shared upstream path (the edge-to-origin link between the client-facing cache tier—such as a content delivery network (CDN) point of presence, campus gateway, or wireless backhaul aggregation point—and the content origin), wasting upstream bandwidth, increasing congestion, and degrading startup latency and quality for all users.

Figure 1 illustrates the setting. Users explore different regions but require common base-layer blocks. We call the coarse lower layers that make a region renderable its *prerequisites*, and the higher layers that add detail its *refinements*. We define the *shared retrieval base* (or *shared base*) as the set of prerequisite and closure blocks the proxy

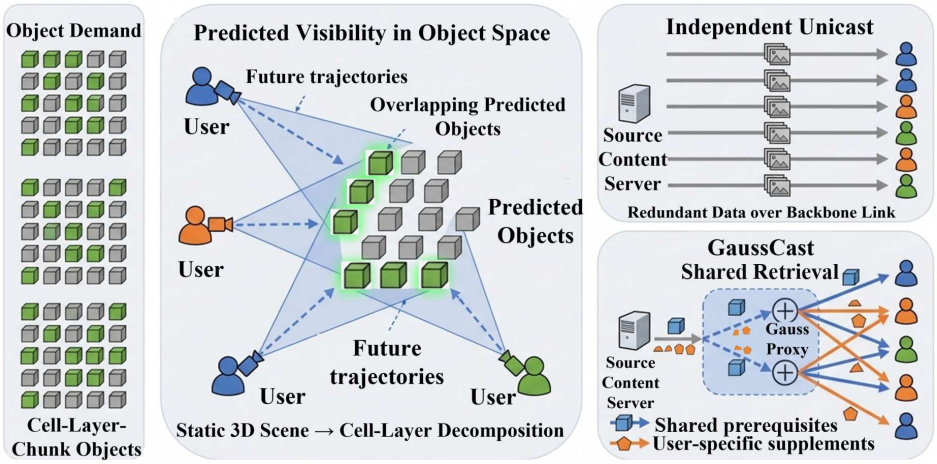


Fig. 1 Multi-user scenario: concurrent users explore different viewpoints but depend on overlapping lower-layer blocks. GaussCast fetches shared prerequisites once and adds per-user supplements.

selects in the shared planning stage for upstream fetch-once reuse across users in the current planning window; the edge proxy fetches this set once over the upstream path and reuses it across those users, in contrast to *per-user supplements*, which are the remaining blocks a user needs on its own—predominantly refinements, together with any prerequisites that only that single user still requires. Fetching the shared base once, then adding per-user supplements, reduces upstream traffic and improves cache utilization. Standard edge caches and request aggregation can partially alleviate the problem, but do not coordinate which blocks to fetch jointly. More critically, they ignore rendering dependencies: a refinement that arrives on time but ahead of its required lower layers is still unusable, wasting bandwidth without improving rendered quality.

GaussCast addresses these challenges through dependency-aware shared retrieval. It does not propose a new codec or renderer; instead, it operates on existing layered 3DGS representations and coordinates their delivery across users. Key design principles include: (1) structuring scenes as versioned, named retrieval blocks with explicit dependency metadata, (2) aggregating short-horizon user demand at an edge proxy to identify reusable shared prerequisites, (3) enforcing dependency closure during admission—the set of prerequisite blocks a refinement depends on—so a block is scheduled only when its missing dependencies fit the budget and meet rendering deadlines, and (4) using digest-linked verification to keep per-block cryptographic overhead practical. These principles enable the proxy to build a shared retrieval base of reusable prerequisites fetched once upstream, then allocate remaining budget to user-specific supplements, while ensuring that admitted content is both dependency-complete and renderable before its deadline. The core of GaussCast is this planner. ICN naming, digest-linked verification, cache retention, and optional parameter tuning are implementation mechanisms that support the planner but are not required for the abstraction.

This paper makes the following contributions:

Table 1 Target and non-target deployment scenarios for GaussCast.

Case	Delivery condition	Expected behavior
Small static scene	Compressed scene fits startup budget	Use pre-download
Single isolated user	No concurrent overlap	Falls back to per-user delivery
Shared virtual tour	Users follow similar paths	Shared base reduces traffic
Collaborative inspection	Users examine related regions	Edge cache reuses prerequisites
Updated digital twin	Regions change over time	Versioned blocks avoid full re-transfer
Mobile XR	Bandwidth varies, cold caches	Progressive retrieval reduces startup

1. We formulate multi-user layered 3DGS delivery as a dependency-aware shared block retrieval problem, separating how a scene is compressed from how its blocks are scheduled, shared, and cached across concurrent users.
2. We design a versioned retrieval-block representation that keeps three concerns independent: naming structure for transport, rendering-dependency graph for usefulness, and digest-linked structure for verification. This separation allows GaussCast to work with any layered 3DGS pipeline without requiring changes to the codec or renderer.
3. We develop a closure-aware planner that builds a shared base and per-user supplements under cache, deadline, byte-budget, and fairness constraints. The planner admits a block only with its missing prerequisites, and weights view-dependent value per user, so that admitted content is both renderable before its deadline and relevant to the user's viewpoint. We prove that admitted blocks satisfy dependency completeness, and add optional online parameter tuning.
4. We implement and evaluate GaussCast with layered 3DGS scenes and multi-user sessions constructed from real six-degrees-of-freedom (6DoF) traces. Compared with per-user delivery, GaussCast reduces shared edge-to-origin upstream traffic by 26%, lowers time-to-first-frame by 27%, improves viewport quality by 0.97 dB PSNR, and improves fairness, with gains growing as cross-user overlap increases.

2 Problem Scope

GaussCast addresses scenarios where three conditions hold: (1) multiple users access the same scene version with sufficient lower-layer overlap, (2) full pre-download is infeasible due to scene scale, bandwidth limits, or sub-second startup targets, and (3) users join with cold caches or access updated scene versions. Table 1 characterizes target and non-target deployments.

The benefit depends on measurable overlap. Before comparing delivery policies, we therefore characterize the workload itself: across the multi-user windows constructed from real 6DoF traces, concurrent users share a substantial fraction of lower-layer

blocks (Section 7), which shows that the reuse opportunity GaussCast targets is present rather than assumed. When overlap is low, GaussCast allocates a small shared budget and prioritizes per-user supplements rather than forcing counterproductive sharing. GaussCast operates at delivery time above the codec and renderer. Compression reduces per-block size; shared retrieval reduces how many times each compressed block crosses the shared upstream path when users join concurrently or when updates invalidate cached content. CDN and Information-Centric Networking (ICN) caches remain effective; GaussCast adds rendering-aware priorities and dependency enforcement.

3 Related Work

3.1 3DGS Representation and Compression

3DGS represents scenes with explicit Gaussian primitives for real-time view synthesis [1]. Subsequent work improves its structural organization and scales it to larger scenes. Scaffold-GS uses anchor-guided organization [2]; Octree-GS and hierarchical 3DGS provide level-of-detail hierarchies [3, 5]; CityGaussian and related systems handle large-scale scenes through spatial partitioning and distributed training [4, 15, 16]. These methods provide the spatial and hierarchical structure that delivery systems build upon.

A parallel line of work reduces storage and transfer cost. LightGaussian applies pruning, distillation, and quantization [6]; Compressed 3D Gaussian Splatting uses sensitivity-aware clustering [7]; HAC and HAC++ achieve strong compression through context modeling and entropy coding, reducing certain scenes below 10 MB [8, 10]; and EAGLES targets compact representations with lightweight encoding [9]. Progressive compressed bitstreams further enable on-demand partial access [17], and recent surveys review this rapidly growing literature [11]. These advances make individual scene blocks smaller, but they do not address how blocks are scheduled, shared, and cached when many users access the same scene. GaussCast is complementary: it assumes blocks are already compressed and coordinates their delivery across concurrent users. For small scenes whose full compressed model fits the startup budget, pre-download remains simpler; GaussCast targets cases where scene scale, bandwidth constraints, or strict latency requirements make progressive multi-user delivery necessary.

3.2 Progressive and Adaptive 3DGS Delivery

Progressive 3DGS delivery systems are closely related. LapisGS builds cumulative layered 3DGS for adaptive streaming and progressive rendering [13]. L3GS creates layered representations and scheduling algorithms for bandwidth-adaptive delivery [12]. LTS maps multi-layer 3DGS to DASH-style streaming [14]. Recent work studies viewport prediction, bitrate adaptation, saliency-aware tiling, and dynamic 3DGS video delivery [18–20]. These systems demonstrate that progressive delivery and adaptive scheduling improve single-user experience. When several users access the same scene concurrently, however, two constraints appear that single-user schedulers were not

designed to handle. First, concurrent users request overlapping lower-layer prerequisites at nearly the same time, so independent per-user scheduling repeatedly transfers the same blocks over the shared upstream path. Second, a refinement carries no rendering value until its prerequisites arrive, so admitting it early wastes bandwidth without improving quality. GaussCast addresses this coordination problem with dependency-aware planning: a block is admitted only when its missing prerequisites fit the budget and meet rendering deadlines for all users who need it.

3.3 Multi-User Immersive Media Delivery

Multi-user immersive systems exploit overlap, caching, and multicast. For 360-degree video, tile-based streaming and edge caching exploit viewport overlap [21–24]. For point clouds and volumetric video, systems study adaptive delivery, edge computation, multiview transcoding, and multi-user sharing [25–28], as recent surveys of the volumetric pipeline summarize [29]. These works demonstrate that multi-user content creates shared delivery opportunities and that independent per-user scheduling scales poorly. However, they do not enforce rendering dependencies: a 360° tile or point-cloud segment carries visual value as soon as it arrives, whereas a 3DGS refinement block remains unusable until its prerequisite lower layers are present. Layered 3DGS differs in two ways that make dependency-unaware sharing insufficient. First, the rendering value of a 3DGS block is view-dependent: the same block contributes differently to different viewpoints through projected footprint, depth ordering, and alpha compositing, unlike a 360-degree tile whose contribution to a covering viewport is fixed. Second, a block becomes useful only when its lower-layer prerequisites are already available, so a block cannot be treated as an independently useful unit the way a video tile can. GaussCast reflects both properties in its planner: it estimates per-user, view-dependent value when ranking candidates, and it admits a block together with its missing prerequisites as one unit, so that admitted content is both relevant to the user's viewpoint and renderable before its deadline.

3.4 Name-Based Delivery, Caching, and Verification

Information-Centric Networking (ICN), including Named Data Networking (NDN), retrieves content by name and supports in-network caching, request aggregation, and data-centric security [30–33]. GaussCast uses these mechanisms because retrieval blocks are immutable within a scene version and can be fetched by name. Interest aggregation through the Pending Interest Table (PIT) records downstream interfaces for outstanding Interests with the same name, forwards one Interest upstream, and returns the arriving Data to all recorded interfaces [31, 34]. This mechanism benefits GaussCast when several users, planned in the same cycle, request the same Data name before the first copy returns: the network then performs one upstream fetch instead of many. Interest aggregation and in-network caching, however, only decide how to satisfy requests that have already been issued. They do not decide which blocks a group of users should request together, whether a block's lower-layer prerequisites have been scheduled, or whether a refinement can still arrive before its rendering deadline. These are the decisions GaussCast makes at the edge proxy, above the

Table 2 Core notation.

Symbol	Meaning
$\mathcal{R}$	Set of retrieval blocks
$r = (c, l, k)$	Cell–layer–chunk block
$s(r)$	Compressed block size
$D(r)$	Hard dependencies of r
$\mathrm{Cl}(r)$	Dependency closure of r
$\mathcal{A}_x$	Active candidate set at proxy x
U_x	Users below proxy x
C_x	Valid cache state at proxy x
L_u	Blocks already cached at client u
W_x	Upstream byte budget in one cycle
B_x	Shared base selected by proxy x
$B_{x,u}$	Shared-base blocks referenced by user u
$S_{x,u}$	Supplement blocks for user u
Π_u	Plan of user u in the current cycle ($B_{x,u} \cup S_{x,u}$)
G_x	Blocks already included in the current-cycle plan
$p_u(r)$	Short-horizon demand score
$g_u(r)$	User-specific value
$a_x(r)$	Cross-user sharing signal
$t_u^{\mathrm{ddl}}(r)$	Useful deadline of r for user u
κ_t	Controller output at cycle t

forwarding plane. The same caching and reuse behavior can be approximated with HTTP object caches [35], CDN caches, or application relays, although these substrates differ from ICN in how they collapse concurrent requests; we detail the mapping and its semantic differences in Section 6.

Block-level retrieval requires efficient verification. If each small block needed independent public-key verification, the cryptographic cost would outweigh sharing gains. File-Like ICN Collections use signed manifests, hash links, and manifest trees to organize and authenticate large collections [36], and embedded manifests let a consumer fetch and verify a large named object through a hash-linked structure rather than per-block signatures [37]. Scalable content-integrity protocols for ICN extend this idea to high request rates [38]. GaussCast follows the same principle: the publisher signs roots or subtree roots, while block contents are verified through digest links. The proxy may provide hints that shorten traversal but is never a trust root.

4 System Model

Table 2 summarizes the core notation used throughout this section and the design.

4.1 Retrieval Blocks and Dependencies

We consider a static or slowly changing layered 3DGS scene published under a versioned namespace. Within one version, retrieval block names and content are

immutable. Updates create new versions or delta manifests. The scene is decomposed into retrieval blocks $\mathcal{R} = \{r_1, r_2, \ldots, r_N\}$. We denote each block $r \in \mathcal{R}$ as a tuple $r = (c, l, k)$, where c is a spatial cell identifier, l is a representation layer (indexed from 0), and k is a chunk index (indexed from 1). Each block has size $s(r)$, a content digest, and a hard dependency set $D(r)$. A retrieval block is a transport, cache, and scheduling unit; it does not correspond to a semantic object in the reconstructed scene.

Layered 3DGS representations require lower layers to be decoded before higher layers can be rendered. Progressive delivery must therefore track block dependencies: a refinement layer provides no rendering value until its prerequisite base layers arrive. The dependency closure of block r, denoted $\mathrm{Cl}(r)$, is the smallest set containing r and recursively containing all hard dependencies:

$$r \in \mathrm{Cl}(r), \qquad q \in \mathrm{Cl}(r) \Rightarrow D(q) \subseteq \mathrm{Cl}(r). \tag{1}$$

A block contributes to rendering only if all blocks in its closure arrive before its useful deadline. The planner can use any dependency DAG exposed by the layered representation. Our prototype uses cumulative layered 3DGS with conservative full-prefix dependency within each cell:

$$D(c, l, k) = \{(c, l', k') \mid 0 \le l' < l,\ 1 \le k' \le K(c, l')\}, \tag{2}$$

where $K(c, l')$ is the number of chunks in cell c at layer l'. If a source pipeline exposes finer dependencies, the planner can use them directly. We require the dependency graph for one scene version to be acyclic; temporal updates are represented by new versions or delta manifests rather than cycles.

4.2 Users, Cache, and One-Cycle Objective

At edge proxy x, let U_x be the downstream users, $C_x \subseteq \mathcal{R}$ be the verified edge-cache state, and W_x be the upstream byte budget for the current cycle. Each client u reports a compact summary of its local cache L_u. Blocks in L_u are already available locally and removed from that user's active demand. Blocks in C_x are available at the edge; they may appear in a user's plan if the client lacks them, but they consume no upstream bytes. Feasibility is tested on user–block pairs rather than block identifiers alone: the same upstream object can be shared, while the set of blocks referenced or delivered downstream remains user-specific. For each active candidate, the proxy estimates value $g_u(r)$ and useful deadline $t_u^{\mathrm{ddl}}(r)$. The useful deadline is the earliest predicted time when block r is required to render user u's view within the prediction horizon, derived from pose prediction and frame scheduling. A block arriving after this deadline provides no rendering value for the corresponding frame.

The proxy selects shared base B_x and user-specific supplement sets $\{S_{x,u}\}_{u \in U_x}$. Recall that a *refinement* is a content-level notion—a higher-layer block that adds detail on top of its lower-layer prerequisites—whereas a *supplement* is a delivery-level notion—the per-user block set $S_{x,u}$ scheduled after the shared base. A supplement is predominantly composed of refinements, but may also contain a prerequisite that only a single user still needs; the two terms are therefore related but not interchangeable.

The shared base is a set of block references chosen during the shared stage, not a mandatory broadcast. It may include blocks already cached at the edge. For each user, $B_{x,u} \subseteq B_x$ denotes the shared-base blocks in that user's plan. User u receives or reuses:

$$\Pi_u \triangleq B_{x,u} \cup S_{x,u}. \tag{3}$$

Realized utility is:

$$Q_u(\Pi) = \sum_{r \in \Pi_u} g_u(r) z_u(r; \Pi), \tag{4}$$

where $z_u(r; \Pi) = 1$ only if every block in $\mathrm{Cl}(r) \setminus L_u$ becomes available to user u before $t_u^{\mathrm{ddl}}(r)$, and 0 otherwise. This additive utility is a scheduling score rather than a replacement for rendered-image metrics; evaluation reports PSNR, SSIM, late misses, and useful-byte breakdowns separately. Upstream bytes are charged once over the union of planned blocks not cached at the edge:

$$\mathrm{Bytes}_x(\Pi) = \sum_{r \in \left(\bigcup_{u \in U_x} \Pi_u\right) \setminus C_x} s(r). \tag{5}$$

This models the shared upstream path between publisher, parent cache, or core network and the edge proxy. Downstream delivery to clients remains per-user, reflected through access-rate constraints, deadlines, and measured late misses. If last-hop links are the dominant bottleneck, a downstream byte or airtime term can be added; the dependency-closure test itself is unchanged. Fairness is measured with Jain's index [39] over normalized delivered utility. Let $Q_u^{\max}$ be the total positive value of user u's active candidates in the current window, after removing locally cached blocks. Define normalized utility:

$$\widehat{Q}_u(\Pi) = \frac{Q_u(\Pi)}{Q_u^{\max} + \epsilon}, \tag{6}$$

where $\epsilon > 0$ prevents division by zero. Fairness is:

$$\mathrm{Jain}_x(\Pi) = \frac{\left(\sum_{u \in U_x} \widehat{Q}_u(\Pi)\right)^2}{|U_x| \sum_{u \in U_x} \widehat{Q}_u(\Pi)^2 + \epsilon}. \tag{7}$$

Normalizing by current feasible demand avoids treating low-demand and high-demand users as having equal attainable utility. When no user receives positive utility, fairness is set to 1 so idle cycles are not penalized. The late-miss penalty counts admitted user–block pairs that arrive after their useful deadline or lack complete dependency closures:

$$\mathrm{Late}_x(\Pi) = \sum_{u \in U_x} \sum_{r \in \Pi_u} (1 - z_u(r; \Pi)), \tag{8}$$

where $z_u(r; \Pi)$ is the usefulness indicator from Eq. (4). A compact one-cycle objective is:

$$\max_{\Pi} \sum_{u \in U_x} Q_u(\Pi) - \lambda_{\mathrm{bw}} \mathrm{Bytes}_x(\Pi) - \lambda_{\mathrm{late}} \mathrm{Late}_x(\Pi) - \lambda_{\mathrm{fair}} (1 - \mathrm{Jain}_x(\Pi)), \tag{9}$$

subject to $\text{Bytes}_x(\Pi) \leq W_x$ and link-rate and deadline constraints. Indicator z_u and late-miss term $\text{Late}_x(\Pi)$ encode rendering usefulness under dependencies. Here $\text{Late}_x(\Pi)$ counts admitted user–block pairs missing deadline or lacking complete closures, and λ weights set tradeoffs among utility, upstream bytes, lateness, and fairness. Solving this exactly online would require searching many candidate subsets with shared costs, user-specific values, and coupled feasibility through dependencies. Because block values interact through shared closures and upstream costs, GaussCast does not claim constant-factor approximation. Instead, GaussCast uses a two-stage online planner: first build a shared base, then allocate remaining budget to user-specific supplements. The objective defines tradeoffs; evaluation reports individual terms rather than relying on a single weighted score.

5 GaussCast Design

5.1 Overview

GaussCast separates edge planning from data transfer. Edge planning uses visibility, layer priority, dependencies, deadlines, cache state, and fairness. The retrieval path fetches named blocks, aggregates repeated requests when supported, caches returned data, and verifies content. Figure 2 shows the end-to-end workflow: the publisher produces versioned retrieval blocks and dependency metadata offline; the edge proxy receives per-user demand summaries, computes a shared base plus user-specific supplements (the per-user block sets scheduled after the shared stage, predominantly refinements together with any prerequisites only that user needs) under a closure-aware admission test, and issues a retrieval plan; the retrieval path fetches the named blocks, collapses concurrent requests for the same block, fills the edge cache, and verifies content before rendering. The central mechanism is the two-stage closure-aware planner illustrated in Figure 3; the publishing, naming, and verification machinery exists to support the planner. The remainder of this section follows these three stages in order.

5.2 Offline Publishing

The publisher starts from a layered 3DGS scene produced by an existing pipeline [12–14]. The publisher partitions the scene into cells, layers, and chunks. In our prototype, a retrieval block typically contains 0.5–2 MB of coded Gaussian data. Large blocks can be segmented for transfer while scheduling and caching remain at block granularity. GaussCast separates three structures. The naming structure identifies retrieval blocks and manifests. The rendering dependency graph defines when a block becomes useful. The authentication structure defines how a block is verified. This separation keeps the retrieval path simple: the retrieval path fetches named bytes, while the proxy reasons about rendering prerequisites. A representative namespace is:

/gc/scene/⟨scid⟩/ver/⟨v⟩/root/⟨digest⟩
/gc/scene/⟨scid⟩/ver/⟨v⟩/cell/⟨cid⟩/mf/⟨digest⟩
/gc/scene/⟨scid⟩/ver/⟨v⟩/cell/⟨cid⟩/layer/⟨l⟩/chunk/⟨k⟩/seg/⟨s⟩/⟨digest⟩
/gc/ctl/proxy/⟨gid⟩/session/⟨sid⟩/plan/⟨seq⟩/⟨param-digest⟩

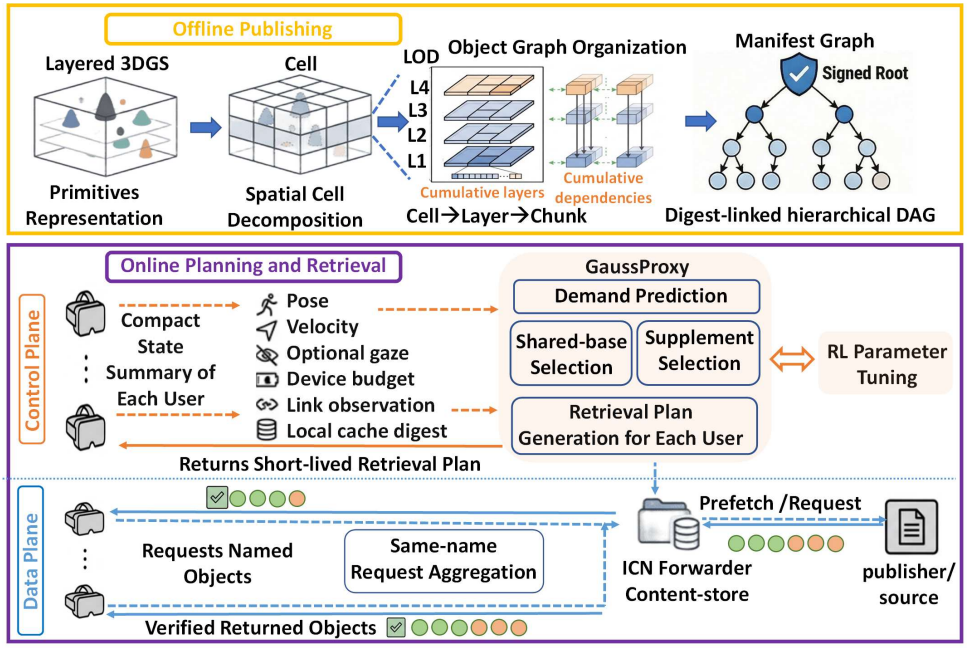


Fig. 2 GaussCast workflow. The publisher builds versioned retrieval blocks; the edge proxy plans a shared base and per-user supplements; the retrieval path fetches, caches, aggregates, and verifies blocks.

The `ver` field separates scene revisions; `root` and `cell/mf` name the signed scene root and per-cell manifests; `layer/chunk/seg` names an individual data segment; and the `ctl` namespace carries short-lived per-session retrieval plans. The trailing `<digest>` binds every name to its content, so a cached copy can be verified without contacting the publisher.

5.3 Demand and Value Estimation

Let $y_{u,h}$ be the predicted pose of user u at future step h, for $h = 1, \ldots, H$, where H is the prediction horizon length. The prototype obtains $y_{u,h}$ from a lightweight constant-velocity predictor over recent head motion, refined by the reported gaze direction when available. GaussCast treats the predictor as a replaceable module and depends only on the resulting pose estimates. For block r, the proxy computes soft visibility score $\tilde{v}(r, y_{u,h}) \in [0, 1]$ from cell bounds, frustum overlap, distance, viewing direction, and optional gaze. Short-horizon demand score is:

$$p_u(r) = 1 - \prod_{h=1}^{H} (1 - \tilde{v}(r, y_{u,h})). \tag{10}$$

If visibility scores are calibrated probabilities, $p_u(r)$ is the probability that user u needs block r at least once within the prediction window; otherwise it serves as a consistent

priority score in $[0, 1]$. User-specific value is:

$$g_u(r) = p_u(r) I(r) w_u^{\text{view}}(r) w_u^{\text{dev}}(r), \tag{11}$$

where $I(r)$ is a layer utility prior, $w_u^{\text{view}}(r)$ measures view relevance, and $w_u^{\text{dev}}(r)$ adapts for device and link conditions. The two viewpoint-dependent factors play different roles: $p_u(r)$ is a temporal term, the probability (or priority) that block r is visible at *some* step within the horizon, obtained by combining per-step visibility across $h = 1, \ldots, H$ in Eq. (10), whereas $w_u^{\text{view}}(r)$ is an angular term that grades *how central* the block is to the predicted view at its first-visibility pose. A block can be near-certain to enter the view yet sit at the periphery (high p_u, lower w_u^{view}), or be foveal but only briefly visible (lower p_u, high w_u^{view}). Concretely, $I(r)$ is non-increasing in layer index l, set from marginal quality gain that the block's layer contributes over prerequisites, normalized to $(0, 1]$; in the prototype $I(r)$ is calibrated from per-layer PSNR improvement measured offline. Factor $w_u^{\text{view}}(r) \in (0, 1]$ increases with frustum overlap and decreases with angular distance to predicted forward or gaze direction. Factor $w_u^{\text{dev}}(r) \in (0, 1]$ scales value by display resolution, decode budget, and sustained link rate so unusable detail is down-weighted. Useful deadline $t_u^{\text{ddl}}(r)$ is the predicted first-visibility time of r within the horizon, as defined in Section 4. For sharing, the proxy computes aggregate demand signals:

$$\phi_x(r) = \sum_{u \in U_x} p_u(r), \qquad \rho_x(r) = 1 - \prod_{u \in U_x} (1 - p_u(r)). \tag{12}$$

When $p_u(r)$ is calibrated, $\phi_x(r)$ is the expected number of users requesting the block and $\rho_x(r)$ is the probability at least one user needs it. Otherwise, they serve as bounded aggregate scores. The sharing signal is:

$$a_x(r) = \max(0, \phi_x(r) - \rho_x(r)). \tag{13}$$

Since $\phi_x(r) \geq \rho_x(r)$ in exact arithmetic, the max protects against numerical error. Intuitively, $\phi_x(r)$ counts expected per-user transfers, while $\rho_x(r)$ represents one upstream fetch cost; their difference estimates avoided duplication.

5.4 Shared-Base Score

The proxy scores only active candidate set $\mathcal{A}_x$, built each cycle from frustum filtering, predicted motion, layer limits, and cache state; blocks outside the prediction horizon are excluded. [1] For each candidate block, the likely user set is:

$$U_x(r) = \{u \in U_x \mid p_u(r) \geq p_{\min}\}. \tag{14}$$

[1] The planning description here focuses on scoring and admission; the construction of $\mathcal{A}_x$ and related security checks are detailed in Appendix C.

Candidates with an empty likely user set are skipped. The threshold prevents negligible nonzero demand from dominating deadline tests. The earliest relevant deadline is:

$$t_x^{\mathrm{ddl}}(r) = \min_{u \in U_x(r)} t_u^{\mathrm{ddl}}(r). \tag{15}$$

This single deadline serves as a fast conservative pre-filter discarding clearly infeasible candidates; admission then applies per-user tests. The shared stage ranks candidate blocks with:

$$S_x(r) = \eta_1 a_x(r) \bar{g}_x(r) + \eta_2 \mathrm{cent}(r) - \eta_3 \mathrm{cost}_x(r) - \eta_4 \mathrm{cong}_x(r), \tag{16}$$

where $\bar{g}_x(r)$ is mean value over $U_x(r)$, $\mathrm{cent}(r)$ measures how often r appears in other blocks' closures, $\mathrm{cost}_x(r)$ is uncached byte cost, and $\mathrm{cong}_x(r)$ penalizes expected congestion. All terms are normalized over active candidates. Overlap weight η_1 is fixed in default configuration, while η_2–η_4 may be adjusted by the optional controller. High score makes a block a candidate; admission still requires closure and deadline tests.

5.5 Closure-Aware Shared-Base Planning

Let A_u^{map} be shared-base blocks already assigned to user u in the current cycle, and let $B_x = \bigcup_u A_u^{\mathrm{map}}$ be their union. A shared candidate is selected for likely user set $U_x(r)$, but not every user needs every prerequisite because local caches differ. The shared stage uses a user-indexed missing closure:

$$\mathrm{MCl}_x^{\mathrm{sh}}(r \mid A^{\mathrm{map}}) = \bigcup_{u \in U_x(r)} \left(\mathrm{Cl}(r) \setminus (L_u \cup A_u^{\mathrm{map}}) \right). \tag{17}$$

This set contains every block that at least one likely user still needs for target block r to become renderable. A block already cached at the edge or already planned in B_x has zero additional upstream cost but may still be added as a reference to a new user's plan. Marginal upstream cost is:

$$\mathrm{Cost}_x^{\mathrm{sh}}(M \mid B_x) = \sum_{q \in M \setminus (C_x \cup B_x)} s(q). \tag{18}$$

Shared-stage density is:

$$\delta_x^{\mathrm{sh}}(r \mid A^{\mathrm{map}}) = \frac{S_x(r)}{\mathrm{Cost}_x^{\mathrm{sh}}(\mathrm{MCl}_x^{\mathrm{sh}}(r \mid A^{\mathrm{map}}) \mid B_x) + \epsilon}. \tag{19}$$

Because the shared-base score $S_x(r)$ in Eq. (16) subtracts cost and congestion terms and can turn negative, density is compared only among candidates that pass the positive-marginal-value feasibility test below, so a negative-score block is never ranked or admitted. A candidate is feasible only if the uncached and unplanned part of its missing closure fits the remaining shared budget, has positive marginal value after costs, and is expected to arrive before relevant user deadlines. For missing closure M

and already scheduled set G, the proxy estimates the completion time of all bytes in M as:

$$\widehat{T}_x(M \mid G) = t_0 + \widehat{\mathrm{RTT}}_x + \frac{\widehat{\beta}_x^0 + \sum_{q \in (G \cup M) \setminus C_x} s(q)}{\widehat{R}_x}, \tag{20}$$

where t_0 is current cycle planning time, $\widehat{\beta}_x^0$ is estimated upstream backlog at cycle start, $\widehat{R}_x$ is available upstream rate, and $\widehat{\mathrm{RTT}}_x$ is round-trip estimate. In the shared stage $G = B_x$; in the supplement stage $G = G_x$. The proxy checks every $u \in U_x(r)$ whose closure $\mathrm{Cl}(r) \setminus L_u$ is not yet fully covered by A_u^{map} (the *affected users*) and admits the candidate only when $\widehat{T}_x(M \mid G) \le t_u^{\mathrm{ddl}}(r)$ for all affected users. If feasible and selected, the required missing closure is admitted as one unit. Blocks in the closure are assigned only to users that still need them, while upstream fetch is charged once over the union of uncached and unplanned block names. This rule prevents the planner from spending bandwidth on refinements valuable in isolation but unusable before prerequisites arrive. It also avoids an artifact of simpler shared-block formulations: a prerequisite one client holds locally should not force an unnecessary downstream reference for that client, yet the same prerequisite may still be worth fetching once for other users. The following proposition makes the resulting guarantee explicit.

Proposition 1 (Shared-stage closure feasibility) *Fix one planning cycle and assume (i) the hard-dependency graph of the scene version is acyclic and $D(\cdot)$ lists every block required for rendering, and (ii) every block in C_x and in each L_u is valid for the requested scene version. Then for every target block r admitted in the shared stage and every affected user u:*

(a) (Dependency completeness, unconditional.) *Every block of $\mathrm{Cl}(r) \setminus L_u$ is recorded in u's plan for this cycle.*

(b) (Timeliness, conditional.) *If additionally (iii) admitted closures are fetched in admission order without preemption and Eq. (20) upper-bounds the realized arrival time of the last byte of the planned set, then every such block is available at the edge no later than $t_u^{\mathrm{ddl}}(r)$.*

Proof (a) When r is admitted, the planner inserts the full missing closure $M = \mathrm{MCl}_x^{\mathrm{sh}}(r \mid A^{\mathrm{map}})$ as one unit, so by Eq. (17), $\mathrm{Cl}(r) \subseteq L_u \cup A_u^{\mathrm{map}} \cup M$ for each affected user u; every block of $\mathrm{Cl}(r) \setminus L_u$ thus lies in $A_u^{\mathrm{map}} \cup M$ and is recorded in u's plan. This step uses only (i)–(ii) and the admit-as-one-unit rule, so it holds regardless of network timing. *(b)* Blocks of $(A_u^{\mathrm{map}} \cup M) \cap C_x$ are already at the edge. The remaining blocks lie in $(G \cup M) \setminus C_x$ with $G = B_x \supseteq A_u^{\mathrm{map}}$. Because $\widehat{T}_x(\cdot \mid G)$ is monotone non-decreasing in the planned set and admission order is preserved without preemption, no block admitted earlier is delayed by a later admission; by (iii) every remaining block then arrives no later than $\widehat{T}_x(M \mid G)$. The admission test requires $\widehat{T}_x(M \mid G) \le t_u^{\mathrm{ddl}}(r)$ for every affected user, which gives the claim. □

The supplement stage admits blocks by the same rule—it inserts the full still-missing closure $\mathrm{MCl}_{x,u}^{\mathrm{sup}}(r)$ as one unit and applies the identical completion-time test (Eq. (20) with $G = G_x$). Since the supplement stage only appends to the plan and never reorders earlier shared admissions, the monotonicity argument of part (b) carries over, so both parts of Proposition 1 extend to every block in the final per-user plan

Π_u. Part (a) is what the planner guarantees by construction; part (b) is the part that prediction and rate estimates can violate in practice, and such late or incomplete closures are counted by the late-miss metric in the evaluation. Figure 3 summarizes the two-stage admission flow.

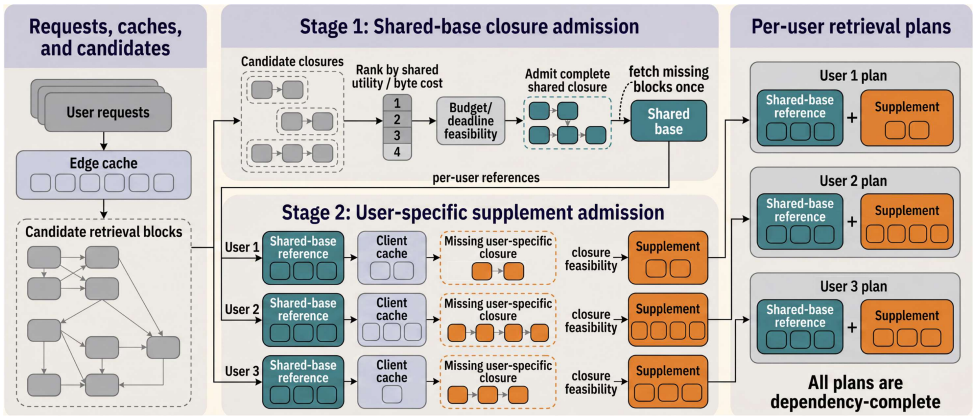


Fig. 3 Two-stage retrieval planning in GaussCast: the edge first fetches common prerequisite blocks once as a shared base, then adds per-user supplements subject to budget, deadline, and dependency checks.

5.6 User-Specific Supplements and Fairness

After the shared stage, the remaining upstream budget, including any unused shared budget, is used for per-user supplements. Recall from Section 4 that $\Pi_u = B_{x,u} \cup S_{x,u}$ is user u's plan; during the supplement stage this plan grows incrementally. Let

$$G_x = B_x \cup \bigcup_{v \in U_x} S_{x,v} \tag{21}$$

denote all blocks already included in the current cycle's plan, across all users. For user u, the closure that must still be added to that user's plan is

$$\mathrm{MCl}^{\mathrm{sup}}_{x,u}(r) = \mathrm{Cl}(r) \setminus (\Pi_u \cup L_u)\,. \tag{22}$$

A block that is already in another user's plan has zero additional upstream cost, but it may still need to be referenced in user u's plan. A block that is already in L_u is neither fetched nor referenced for that user. If an already planned block is in B_x, it is added to $B_{x,u}$; otherwise it is added to $S_{x,u}$ without charging upstream bytes again. The upstream byte cost of adding a supplement closure is

$$\mathrm{NewBytes}_x(M \mid G_x) = \sum_{q \in M \setminus (C_x \cup G_x)} s(q). \tag{23}$$

The supplement density is

$$\delta^{\mathrm{sup}}_{x,u}(r) = \frac{\omega^{\mathrm{fair}}_u \omega^{\mathrm{urg}}_{u,r} g_u(r)}{\mathrm{NewBytes}_x(\mathrm{MCl}^{\mathrm{sup}}_{x,u}(r) \mid G_x) + \epsilon}. \tag{24}$$

Here ω^{fair}_u gives bounded priority to users with lower recent quality of experience (QoE), and $\omega^{\mathrm{urg}}_{u,r}$ increases priority as the deadline approaches. Both weights are clipped to avoid oscillation. User selection uses fair rotation so that a user with many high-value candidates cannot consume the entire supplement budget. The planner may reject a higher-layer block even when its isolated value is high if the missing required lower layers would exceed the current deadline or budget. This policy may skip some visual detail under perfect prediction, but it avoids spending the upstream budget on refinements that cannot be used for rendering. When overlap is low, dependency closures are large, or deadlines are short, the plan shifts budget toward per-user supplements. In one cycle, the dominant cost is sorting the active candidate set and evaluating cached closure summaries; the active set is bounded by the prediction horizon rather than by the full scene. With $|\mathcal{A}_x|$ active candidates and average closure size $\bar{d}$, one planning cycle costs $\mathcal{O}(|\mathcal{A}_x| \log |\mathcal{A}_x|)$ for ranking plus $\mathcal{O}(|\mathcal{A}_x|\, \bar{d}\, |U_x|)$ for the per-user missing-closure and deadline tests, both independent of the full scene size N.

5.7 Optional Parameter Controller

GaussCast does not rely on learning: the default system runs the deterministic closure-aware planner with fixed parameters and is dependency-safe on its own. We optionally evaluate a lightweight controller that adjusts only high-level knobs—the shared-budget ratio, the dependency/cost/congestion weights, and the cache-retention and supplement thresholds—as bandwidth, overlap, or cache pressure drifts. Its inputs are aggregate signals (group size, mean lower-layer overlap, cache occupancy, queue backlog, mean RTT and throughput, miss and late ratios, and the fairness index), and it is trained with proximal policy optimization (PPO) [40]; the exact action, state, and reward are given in Appendix B. The controller never selects individual blocks, changes names or digests, bypasses cache-version checks, or admits a block that fails the closure and deadline tests. It is therefore a tuning aid rather than a core contribution, and it adds only a few percent on top of the deterministic planner; we report its effect separately in Section 7.11 and Appendix E.

6 Retrieval, Caching, Verification, and Delivery Mapping

6.1 Plan-Based Retrieval and Request Aggregation

Clients periodically request a plan from the edge proxy and include compact session summaries. A returned plan contains block names, per-user shared-base or supplement tags, priority order, pacing hints, and verification hints. A shared-base tag marks a block whose single upstream fetch can be reused for several users; it does not mean the block is pushed to every client. Each client retrieves only the blocks listed in its

Algorithm 1 One-Cycle Planner at Edge Proxy x

Require: Candidate set $\mathcal{A}_x$, cache C_x, budget W_x, values $g_u(r)$, deadlines $t_u^{\mathrm{ddl}}(r)$, controller parameters κ_t (Section 5.7)

Ensure: Shared base B_x, per-user shared assignments $\{B_{x,u}\}$, and supplements $\{S_{x,u}\}$

$W_x^{\mathrm{sh}} \leftarrow r_t^{\mathrm{sh}} W_x$; $W_x^{\mathrm{sup}} \leftarrow W_x - W_x^{\mathrm{sh}}$ ▷ split budget by the shared ratio in κ_t

Compute $p_u(r)$, $g_u(r)$, $a_x(r)$, and $S_x(r)$ for active candidates

$B_x \leftarrow \varnothing$ and initialize an empty assignment map A^{map}

while $W_x^{\mathrm{sh}} > 0$ and feasible shared candidates with positive value remain **do**

  Select feasible r^* with largest $\delta_x^{\mathrm{sh}}(r^* \mid A^{\mathrm{map}})$

  $M \leftarrow \mathrm{MCl}_x^{\mathrm{sh}}(r^* \mid A^{\mathrm{map}})$

  Deduct $\sum_{q \in M \setminus (C_x \cup B_x)} s(q)$ from W_x^{sh}, then admit M into B_x and record each block only for users in $U_x(r^*)$ that still need it

end while

$W_x^{\mathrm{sup}} \leftarrow W_x^{\mathrm{sup}} + W_x^{\mathrm{sh}}$ ▷ unused shared budget rolls over

$G_x \leftarrow B_x$

for all $u \in U_x$ **do**

  $S_{x,u} \leftarrow \varnothing$

  Initialize $B_{x,u}$ with blocks recorded for u in A^{map}

  Build feasible supplement list R_u^{sup} using deadline and value-threshold tests

  Rank R_u^{sup} by $\delta_{x,u}^{\mathrm{sup}}$

end for

while $W_x^{\mathrm{sup}} > 0$ and some R_u^{sup} is nonempty **do**

  Pick next user u by fair rotation

  Recompute marginal byte cost for top candidates in R_u^{sup}; choose the highest-density feasible candidate r and set $M \leftarrow \mathrm{MCl}_{x,u}^{\mathrm{sup}}(r)$

  Deduct $\mathrm{NewBytes}_x(M \mid G_x)$ from W_x^{sup}

  Add $M \cap B_x$ to $B_{x,u}$ and add $M \setminus B_x$ to $S_{x,u}$

  $G_x \leftarrow G_x \cup M$; remove satisfied candidates and candidates that no longer fit the budget or deadline from affected lists

end while

Update cache-retention priorities

return $(B_x, \{B_{x,u}\}_{u \in U_x}, \{S_{x,u}\}_{u \in U_x})$

own plan and skips blocks already present and valid in its local cache. The benefit of the shared base is on the upstream path: the block crosses the edge-to-origin link once and is then served from the edge cache to whichever clients request it, rather than being fetched from the origin separately for each client.

If the delivery mechanism supports request aggregation, concurrent requests for the same shared-base block can be collapsed into one upstream fetch. In the ICN prototype, this collapse is Interest aggregation through the PIT: the first Interest for a name is forwarded upstream, later downstream Interests with the same name are recorded while the first Interest is pending, and the returned Data satisfies the recorded downstream requests. Because a 0.5–2 MB block is carried as multiple Data

packets with distinct segment names, aggregation operates at segment granularity. A block is shared across users only when their Interests for the same segments coincide while pending. Reuse for users that arrive after Data has returned is provided by the content store rather than by the PIT entry itself. Aggregation complements rather than replaces the planner: it merges identical outstanding names, while the planner decides which prerequisites should be requested together, what to retain for slightly later users, and whether a refinement can meet its deadline. If the delivery mechanism lacks request aggregation, the proxy can still fetch each planned block once and redistribute it through an application-level relay.

6.2 Cache Retention

The cache policy favors reusable prerequisite blocks. It is compatible with standard cache freshness and validation rules; the proxy only changes retention priority among blocks that are valid for the requested scene version and digest. Freshness, version, expiry, and digest checks are applied before retention scoring so that reuse cannot silently mix stale blocks from different scene versions. The retention score combines near-future demand, membership in the shared base, dependency centrality, age, and size, with all terms normalized over the currently cached set:

$$H_x(r) = \beta_1 \rho_x(r) + \beta_2 \mathbf{1}[r \in B_x] + \beta_3 \mathrm{cent}(r) - \beta_4 \mathrm{age}(r) - \beta_5 s(r). \tag{25}$$

The policy is asymmetric: lower-layer prerequisite blocks are kept more aggressively than transient higher-layer refinements. The cache-policy sensitivity study separates this bias from generic least-recently-used retention.

6.3 Digest-Linked Verification

The publisher signs the root manifest and, when useful, subtree roots. Lower-level manifests and data segments are authenticated through digest links or Merkle paths. This avoids one public-key verification per small block. Static digest hints are generated by the publisher, and plan-carried hints are attached by the proxy to reduce manifest traversal. A hint cannot create a new trust path. If a hint is stale, missing, or malicious, the client falls back to standard digest-linked traversal; security is unchanged and only latency can increase. Figure 4 shows the trust boundary.

6.4 Delivery Mapping

GaussCast requires only block names, cacheable retrieval, optional request aggregation, and content verification. Table 3 maps these requirements to several delivery mechanisms; mapping ICN primitives onto existing IP and overlay infrastructure follows established deployment considerations for information-centric networking [41]. In the table, *plan delivery* refers to the control channel that carries each cycle's retrieval plan—the proxy's decision of which blocks to fetch and which user each block is assigned to—from the edge proxy to its clients; it is distinct from the data path that carries block bytes. HTTP deployments can use standard cache metadata and validation for object reuse [35]. Relay deployments can run over HTTP/3 streams when

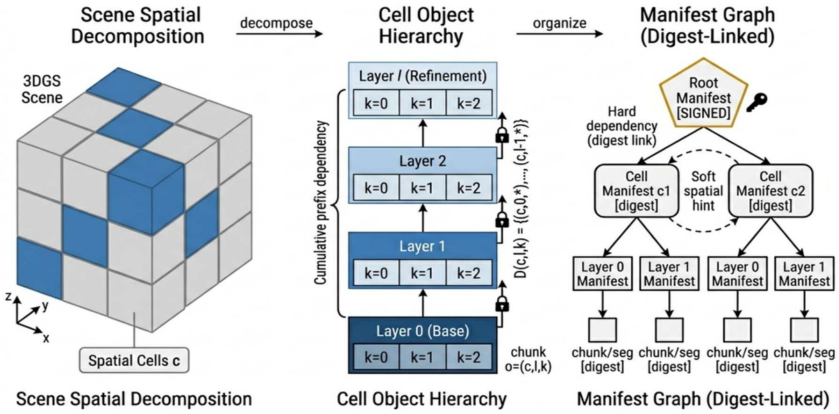


Fig. 4 Block model and digest-linked verification. The scene is partitioned into cell-layer objects with full-prefix dependencies (left, center); a signed root then authenticates lower manifests and data blocks through digest links (right). Hard dependencies (solid) must be retrieved together, whereas spatial hints (dashed) may shorten manifest lookup but cannot authenticate data on their own.

Table 3 Mapping GaussCast primitives to delivery mechanisms.

GaussCast primitive	ICN/NDN	HTTP/CDN	HTTP/3 relay
Block naming	NDN name	URL or object key	object/stream ID
Same-cycle request collapsing	PIT Interest aggregation (native)	request collapsing (if supported by cache)	relay-side aggregation
Edge cache	content store	CDN or object cache	relay cache
Integrity	signed manifest, per-block digest	signed manifest, object hash	signed manifest, hash IDs
Plan delivery	control namespace	dedicated plan endpoint	control stream

low-latency bidirectional communication is desired [42]. The mapping does not imply that all mechanisms offer identical semantics: ICN provides PIT-based Interest aggregation in the forwarding plane, HTTP/CDN deployments rely on cache reuse and need implementation-specific request collapsing or explicit edge aggregation for same-cycle sharing, and relays emulate aggregation at the application layer.

7 Experimental Evaluation

The evaluation answers five questions: (1) does the workload contain reusable lower-layer blocks; (2) does GaussCast reduce upstream traffic, startup delay, and late misses; (3) does closure-aware planning improve useful delivery compared with sharing without closure checks; (4) how do the gains change with overlap, concurrency, compression, dependency sparsity, and prediction horizon; and (5) are planning, control, and verification overheads practical?

Workload characterization first establishes whether sharing opportunity exists; end-to-end results then measure user-visible benefit; ablations separate closure-aware planning from generic caching and request aggregation; sensitivity tests identify when the system should fall back to simpler delivery; and overhead studies check that block-level retrieval is practical. For the main traffic, latency, and quality metrics, we report the mean and standard deviation over all paired samples. Each configuration is replayed over 60 trace windows and 10 network seeds (600 paired samples); unless noted otherwise, the differences between GaussCast and each baseline reported for a given metric (upstream bytes, TTFF, late miss, and PSNR) are statistically significant at $p < 0.01$ under a paired Wilcoxon signed-rank test with Holm correction for multiple comparisons. Because the test pairs GaussCast against each baseline window by window, the reported gains reflect a consistent per-window improvement rather than an average inflated by a few high-overlap windows.

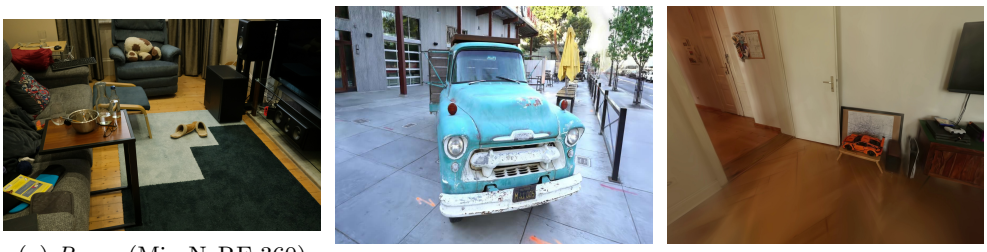
7.1 Prototype and Setup

Prototype.

We implement GaussCast on Mini-NDN [43] with NFD forwarders [44]. The default topology is a three-tier tree with one publisher, one core forwarder, two aggregation forwarders, four edge forwarders (which provide transport and same-name request aggregation but do not partition the planning domain), and clients attached to the edge layer. The edge tier hosts one planning proxy that runs Algorithm 1: in the notation of Section 4, its user set U_x contains all concurrent clients of the session (16 by default), so demand aggregation, shared-base decisions, and the edge-cache state span the full group rather than only the clients attached to a single edge forwarder; the forwarders below the proxy provide transport and same-name request aggregation. Upstream bytes are measured on the shared edge-to-origin link between this cache tier and the publisher. The sharing domain of every reported number is therefore explicit: the upstream reduction comes from planner-level reuse and same-cycle aggregation within this one domain, not from incidental cache overlap between independently planned edges. Deployments that partition clients into several independent planning domains would retain the intra-domain gains and add only whatever reuse their upstream caches provide; we do not evaluate such federation. We also implement an HTTP/CDN baseline in Mininet [45] with the same link capacity, propagation delay, and cache budget to enable fair comparison.

Scenes and traces.

We replay real 6DoF navigation traces from EyeNavGS [46], which records per-frame head pose and eye gaze of 46 participants freely exploring twelve real-world 3DGS scenes from Mip-NeRF 360 [47], Tanks and Temples [48], Deep Blending [49], and Zip-NeRF [50]. Each trace is recorded in a specific scene; traces are replayed only in their captured scene. We select three representative scenes spanning the spatial regimes in Table 1: *Room* (bounded indoor), *Truck* (outdoor object-centric), and *Berlin* (large multi-room indoor—the case where progressive shared delivery matters most). Each scene is converted into layered 3DGS with 3–5 cumulative layers and cell–layer–chunk retrieval blocks; Table 4 summarizes the resulting scene and layer



(a) *Room* (Mip-NeRF 360) (b) *Truck* (Tanks and Temples) (c) *Berlin* (Zip-NeRF)

Fig. 5 Representative evaluation scenes: a bounded indoor scene, an outdoor object-centric scene, and a large multi-room indoor scene, each with real 6DoF traces in EyeNavGS.

characteristics. Multi-user sessions are formed by grouping trace windows of different participants in the same scene with controlled start-time offsets; overlap is reported before performance comparison to avoid selecting only favorable windows. We state this construction explicitly because it bounds how the results should be read: EyeNavGS records each participant exploring a scene *individually*, so the dataset contains no genuinely concurrent multi-user capture. Every multi-user session in this paper is constructed by time-aligning and offsetting independently recorded single-user traces in the same scene, which preserves each user's real per-frame motion but not any real-world correlation between users who happen to share a session. The reported cross-user overlap is a property of how independently captured paths intersect in space, not of coordinated group behavior; the synthetic guided/class-like model (Appendix A) is the one case where we inject inter-user correlation, and it is excluded from every headline number. Consequently, the evaluation should be read as a controlled test of spatial overlap and shared-cache behavior under real 6DoF motion, not as a behavioral study of collaborative XR users. Unless stated otherwise, experiments use 16 clients distributed evenly across four edge forwarders, 200 ms planning interval, 1 s prediction horizon, and 30 fps rendering from the best available complete layered set. Figure 5 shows the three scenes.

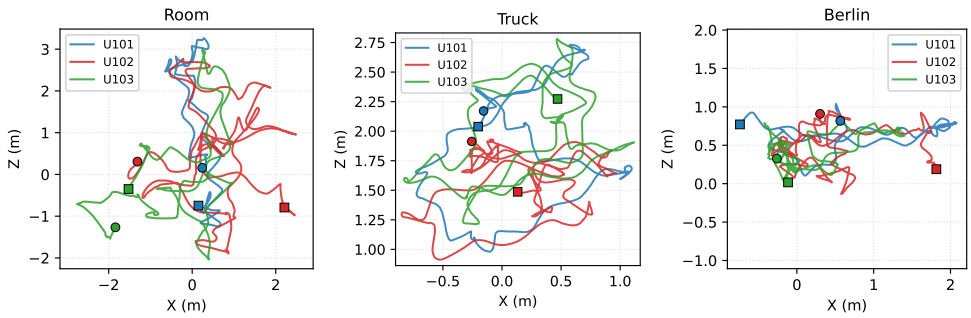
To make the overlap structure concrete, Figure 6 plots the horizontal (XZ-plane) head-position trajectories of three concurrent participants in each of the three evaluation scenes. The paths diverge in the regions each user inspects closely but repeatedly revisit shared areas, which is exactly the lower-layer overlap that the shared base exploits. The EyeNavGS project page[2] provides the full traces and per-user gaze recordings.[3]

Network.

Each client has an access rate of 10, 20, or 30 Mbps. The shared upstream bottleneck is 150, 225, or 300 Mbps. Edge-cache capacity is 20% of the active working set. Results are reported as mean ± standard deviation over scene, trace-window, and

[2] https://symmru.github.io/EyeNavGS/

[3] A multi-user 6DoF playback video is provided as supplementary material (Online Resource 1); it animates six concurrent participants per scene over a 60 s window: each head pose is drawn as a position trail and a view-frustum wedge, with eye-gaze rays overlaid for the scenes whose traces record gaze (e.g. *Room*); overlapping wedges mark the shared lower-layer demand the shared base exploits, making the overlap-and-separation structure visible over time.



(a) *Room* (b) *Truck* (c) *Berlin*

Fig. 6 Horizontal (XZ-plane) head-movement trajectories of three concurrent users per scene from EyeNavGS traces. Paths diverge in closely inspected regions but share common areas, producing the lower-layer overlap the shared base reuses.

network-seed combinations. Table 5 lists the full network and replay configuration. This configuration models typical CDN edge scenarios where multiple users share an upstream path to the origin with limited edge cache capacity.

Rendering and quality measurement.

The PSNR, SSIM, and LPIPS values in Table 9 are measured on rendered viewport frames, not inferred from delivered bytes. For each user we sample frames along that user's real 6DoF trace; at each sampled pose the viewport is rendered twice with the same LapisGS-style layered 3DGS renderer [13]: once from the *full* layered model—all layers of every visible cell present, that is, the complete converted layered representation and not the original uncompressed training model—which we take as the per-frame *reference image*, and once from the set of blocks that the policy under test has actually delivered and made renderable by that frame's deadline. The reported metrics compare the policy's rendered frame against this reference, so they capture the visual cost of missing or late refinements rather than reconstruction error of the codec itself, which is common to all policies and therefore cancels in the comparison. When a cell's required prerequisites have not yet arrived, the renderer falls back to the highest complete layer prefix available for that cell (the coarse base, or the background floor if even L0 is missing), exactly as a deployed client would; a block that arrives without its prefix contributes nothing and is counted as unusable rather than improving the frame. All policies share this identical renderer, reference, fallback rule, and the same sampled poses and frame count (30 fps over each 60 s window), so the only difference between rows is which blocks each policy manages to deliver in renderable form.

Two factors set the scale of the reported numbers. First, the reference is the full layered model, but under the finite upstream budget all policies operate at a lower level of detail than the full model: no policy delivers every refinement layer within each frame's deadline, so all rows sit in the 28–29 dB range rather than near the codec ceiling. The absolute values therefore reflect the budget-limited operating point shared by all policies, not a weakness of any single one. Second, because every policy renders from the same layered codec and the codec's own quantization error is present

Table 4 Scene and layer characteristics. Compressed sizes use the layered 3DGS encoding of each scene; L0/L1 denotes the two lowest (base) layers.

Scene	Dataset	Type	#Gauss.	Comp. size	#Cells	#Lyr	L0/L1 size
Room	Mip-NeRF 360	indoor	1.6 M	38 MB	96	3	6.1 MB
Truck	Tanks&Temples	outdoor	2.4 M	142 MB	128	4	19.8 MB
Berlin	Zip-NeRF	large indoor	21 M	1.27 GB	384	5	121 MB

Table 5 Network and replay configuration.

Parameter	Default value
Concurrent clients	16
Planning interval	200 ms
Prediction horizon	1 s
Client access rate	10/20/30 Mbps
Shared upstream bottleneck	150/225/300 Mbps
Edge cache capacity	20% of active working set
Trace windows	60 windows, 60 s each
Random seeds	10 network seeds

in every row's reference and test frames alike, this codec error cancels in the render-against-render PSNR comparison, leaving only the differences in which refinement layers each policy manages to deliver complete and on time. The 0.97 dB gain is the detail that dependency-aware delivery converts into on-time, dependency-complete refinements, concentrated on the frames and viewport regions that missing prefixes or late refinements would otherwise leave at a coarser level.

7.2 Workload Characterization

Before comparing delivery policies, we characterize reuse opportunity without filtering windows by performance. Table 6 reports visible-cell overlap, lower-layer block overlap, byte-weighted prerequisite overlap, and duplication factor (total requested prerequisite bytes divided by unique prerequisite bytes; 1.0 means no reuse opportunity). Two findings justify the shared-base approach. First, lower-layer overlap consistently exceeds visible-cell overlap (0.34 vs. 0.29 across all windows), confirming that even when users look at different regions, they share a larger fraction of base-layer prerequisites than their visible cells suggest. Second, the duplication factor averages 3.5 across all windows and rises with concurrency—from 1.4 at two users to 6.4 at sixteen—meaning per-user delivery transfers each prerequisite block several times over the shared upstream path, and increasingly so as more users join. The all-windows mean of 3.5 is lower than the 16-user value because the characterization windows span the full 2–32 user range rather than fixing concurrency at the 16-client default. Overlap decreases as cold-start offsets grow but remains nonzero even at a 30 s offset

Table 6 Unfiltered workload characterization (mean ± std over all trace windows). The guided/class-like row (†) is a synthetic model on real guide paths (Appendix A); the cold-start and concurrency rows are separate cuts along one controlled axis each, not breakdowns of the all-windows mean.

Trace group	Visible-cell Jaccard	L0/L1 overlap	Byte-wt. overlap	Dup. factor
All windows	0.29 ± 0.24	0.34 ± 0.25	0.36 ± 0.25	3.5 ± 0.8
Guided / class-like†	0.37 ± 0.21	0.43 ± 0.22	0.45 ± 0.22	3.7 ± 0.9
Free exploration	0.18 ± 0.16	0.23 ± 0.18	0.24 ± 0.18	2.7 ± 0.6
Cold-start				
offset 0/5/10/30 s	L0/L1 overlap: 0.44 / 0.27 / 0.22 / 0.20			
Concurrency				
2/16/32 users	Dup. factor: 1.4 / 6.4 / 9.1			

(0.20 L0/L1 overlap), because late joiners still need the same base layers as earlier users. Free-exploration windows show the lowest overlap and bound the regime where GaussCast falls back toward per-user delivery.

7.3 Baselines and Metrics

The single-user systems we build on assume one session per origin connection and expose no multi-user demand interface, so running their original codebases under shared delivery would fall back to independent per-user fetching. We instead reimplement their session-centric delivery model on the same substrate as GaussCast, holding the codec, renderer, prediction module, cache budget, and network topology identical so that any measured difference reflects the planning policy rather than implementation details. This is a conservative comparison: where possible, baselines run on the same codec, renderer, prediction module, cache budget, topology, and link traces as GaussCast, and substrate-specific features such as PIT aggregation are isolated through separate HTTP, ICN, no-aggregation, and cache-only variants, so multi-user dependency-aware planning is the only variable in each pairing.

PerUser-HTTP Each user fetches blocks independently through a matched HTTP/CDN stack, following the session-centric delivery model of single-user layered 3DGS systems [12, 14]. Standard cache reuse of completed cacheable objects is allowed, but there is no rendering-aware shared planner and no explicit same-cycle request collapsing.

PerUser-ICN Each user uses named retrieval with per-user planning, isolating the gain from generic PIT aggregation [34] and content-store reuse without shared dependency-aware planning.

DependencyAware-PerUser Each user runs the closure-aware admission test of GaussCast on a per-user basis—a block is admitted only with its still-missing prerequisites—but the proxy performs no cross-user shared planning. On the same

named-retrieval substrate as PerUser-ICN, this isolates the effect of dependency closure alone, and together with PerUser-ICN, SharedGreedy, and GaussCast forms a 2×2 design over the two mechanisms (closure-aware admission and cross-user sharing).
SharedGreedy The proxy aggregates cross-user demand under a common budget and benefits from aggregation and cache reuse, but ranks blocks by aggregated value density without closure-aware admission, in the spirit of overlap-driven multi-user prefetching [27, 28].
GaussCast-noClosure Shared and supplement stages use GaussCast scores but admit blocks individually instead of inserting full missing closures together, isolating the contribution of closure-aware admission.
GaussCast-noAggregation The full planner runs, but same-cycle request aggregation is disabled, isolating the contribution of forwarding-plane collapsing.
GaussCast-cacheOnly The full planner runs over an HTTP/CDN-style cache with standard object reuse but no same-cycle request aggregation, modeling a pragmatic Web-cache deployment.
GaussCast-Full The complete system with closure-aware planning, asymmetric caching, request aggregation, and the learned controller.
OracleSharedPrereq An idealized, non-deployable upper bound rather than a baseline policy: every distinct prerequisite block is charged to the shared upstream link at most once over the whole run, modeling perfect deduplication of prerequisites with hindsight. Its remaining behavior matches GaussCast. It is an upper bound only on the prerequisite-deduplication component, not on end-to-end utility or on all possible scheduling policies, so the gap between GaussCast and this bound shows how much of the achievable prerequisite-sharing saving GaussCast realizes online without future knowledge.

Metrics include upstream bytes, edge-cache hit ratio, time-to-first-frame (TTFF), late-miss ratio, useful bytes, unusable bytes, peak signal-to-noise ratio (PSNR), structural similarity (SSIM), learned perceptual image patch similarity (LPIPS), Jain fairness, planner latency, verification CPU cost, and control overhead. TTFF is measured from session join to the first frame whose required prerequisite closure is complete. Late-miss ratio counts admitted user–block pairs arriving after their useful deadline. Useful-byte accounting is computed after dependency validation rather than from delivered bytes alone. This metric set separates traffic reduction from quality improvement and deadline performance. All ratios are paired against PerUser-HTTP on the same scene, trace window, and network seed, so the reported deviations measure the per-pair difference rather than the larger absolute spread across heterogeneous scenes.

7.4 End-to-End Efficiency and Quality

Table 7 reports the main performance comparison. Unless otherwise stated, traffic and control ratios are normalized to PerUser-HTTP under the same scene, trace window, and network seed; absolute values are reported alongside normalized ratios when available. The comparison uses several metrics rather than one aggregate score. Upstream bytes quantify repeated-transfer reduction, TTFF and late miss quantify deadline

Table 7 End-to-end delivery results. Values are reported as mean ± standard deviation. OracleSharedPrereq (below the rule) is an idealized upper bound in which each distinct prerequisite crosses the shared upstream link at most once over the whole run; it is not a deployable policy and bounds only the prerequisite-deduplication component of the upstream saving (the lower bound of 0.68 remains above zero because each distinct block must cross the link at least once and per-user refinements are not shareable), not end-to-end utility or all possible scheduling policies.

Policy	Upstream bytes ↓	Edge hit ↑	TTFF (s) ↓	Late miss ↓	Jain ↑
PerUser-HTTP	1.00	0.31 ± 0.06	0.75 ± 0.11	0.184 ± 0.031	0.887 ± 0.021
PerUser-ICN	0.93 ± 0.04	0.37 ± 0.06	0.69 ± 0.10	0.166 ± 0.029	0.892 ± 0.021
DependencyAware-PerUser	0.89 ± 0.04	0.38 ± 0.06	0.64 ± 0.09	0.108 ± 0.020	0.901 ± 0.020
SharedGreedy	0.82 ± 0.06	0.42 ± 0.05	0.66 ± 0.10	0.147 ± 0.025	0.898 ± 0.020
GaussCast-Full	$\mathbf{0.74 \pm 0.05}$	$\mathbf{0.50 \pm 0.05}$	$\mathbf{0.55 \pm 0.08}$	$\mathbf{0.096 \pm 0.017}$	$\mathbf{0.926 \pm 0.017}$
OracleSharedPrereq	0.68 ± 0.05	0.55 ± 0.05	0.52 ± 0.08	0.090 ± 0.016	0.930 ± 0.016

behavior, and Jain fairness measures whether the shared base disadvantages users whose views are less common.

Across all evaluated scenes and trace groups, GaussCast achieves the best transport efficiency while also improving startup, useful delivery, and fairness. Relative to PerUser-HTTP, GaussCast reduces normalized upstream bytes to 0.74 (a 26% reduction), lowers TTFF from 0.75 s to 0.55 s (a 27% reduction), raises the edge-cache hit ratio from 0.31 to 0.50, and improves Jain's index from 0.887 to 0.926. These headline figures are for the default 16-client configuration of Table 5; because the benefit grows with concurrency (Section 7.7), the 26% reduction reflects the additional reuse that 16 synchronized clients create: the per-window overlap average is 0.34, which falls in the medium band where the reduction is $\approx 21\%$, but concurrent arrivals within the same planning cycle enable further aggregation beyond what a single overlap value predicts. The GaussCast-Full row includes the optional learned controller; the deterministic planner under fixed parameters already reaches most of this gain (upstream bytes ≈ 0.77, TTFF ≈ 0.58 s), with the controller contributing the remainder (Table 17, where its multipliers are relative to the static-parameter planner, not to PerUser-HTTP), so the headline improvement is not an artifact of the learned component. The 26% figure thus decomposes across three mechanisms: generic request aggregation and cache reuse (PerUser-ICN reaches 0.93 through PIT aggregation and content-store reuse alone, with no shared planner, and Table 10 shows that disabling same-cycle aggregation moves GaussCast from 0.74 to 0.86 while cache reuse alone recovers 0.80), closure-aware shared planning, which contributes the largest single step, and optional PPO tuning, which trims a further $\approx$3.8% (Table 17). The gain over SharedGreedy confirms that dependency-aware planning, not aggregation alone, drives the improvement: both aggregate cross-user demand, but only GaussCast admits prerequisites together with the refinements that need them. As Section 7.7 shows, the gain grows with overlap and concurrency and then saturates as the edge cache and upstream path fill; when overlap is weak, GaussCast approaches per-user behavior rather than forcing low-value sharing.

Two reference points frame these numbers. Table 8 arranges the relevant policies as a 2×2 design over the two mechanisms, and the outcome can be stated compactly: closure alone improves lateness, usefulness, and quality; sharing alone reduces

Table 8 The 2×2 mechanism design behind the ablation. Closure-aware admission governs useful delivery and quality; cross-user sharing governs upstream bytes; GaussCast enables both.

	No cross-user sharing	Cross-user sharing
No closure	PerUser-ICN	SharedGreedy / GC-noClosure
Closure	DependencyAware-PerUser	GaussCast

upstream bytes but can waste refinements; GaussCast combines both and realizes 81% of the ideal prerequisite-deduplication saving. Closure alone (DependencyAware-PerUser) cuts late miss from 0.166 to 0.108 because refinements are no longer delivered ahead of their prerequisites, yet it barely moves upstream bytes (0.93 → 0.89): each user still fetches its own copy of every shared prerequisite. The same mechanism gives DependencyAware-PerUser a slightly lower TTFF than SharedGreedy (0.64 s vs. 0.66 s), since admitting each cell's prerequisite closure as one unit completes the first renderable view sooner, whereas SharedGreedy spreads its budget across widely shared blocks ranked by aggregate value and can leave the first user's view one prerequisite short. Sharing alone (SharedGreedy) does the opposite: it cuts upstream bytes to 0.82 but leaves the unusable fraction high. Combining both, GaussCast reaches 0.74 upstream bytes with a low unusable fraction, so the quality and useful-delivery gains are attributable to closure admission while the upstream saving is attributable to cross-user sharing. OracleSharedPrereq bounds how far that saving can go: perfect hindsight deduplication of every prerequisite reaches 0.68, so GaussCast realizes about 81% of the achievable reduction (0.26 of an ideal 0.32) online, without future knowledge. The bound stays well above zero because each distinct prerequisite must still cross the link at least once and per-user refinements are not shareable.

7.5 Useful Delivery and Visual Quality

The useful-byte decomposition separates useful, late, and unusable bytes. A byte is useful if it contributes to a frame that can be rendered before the deadline. A byte is late if it arrives after its useful deadline. A byte is unusable if it arrives but cannot be rendered because prerequisites are missing. Figure 7 separates whether a policy saves bytes by avoiding duplication or merely shifts bytes into late or unusable transfers.[4]

Naive sharing does not by itself improve useful delivery. SharedGreedy reduces late bytes relative to PerUser, but its unusable fraction rises to 15.2% because it admits overlap-heavy refinements without guaranteeing that their missing prerequisites complete in time. GaussCast avoids this by admitting each block together with its still-missing closure, raising the useful fraction to 86.1% and lowering the unusable fraction to 5.9%. This explains why the quality gap exceeds the byte gap: the benefit of dependency-aware planning is less about transferring fewer bytes and more about converting limited bandwidth into dependency-complete, renderable content.

At matched playback times and bandwidth seeds, Table 9 shows that GaussCast improves viewport quality by 0.97 dB PSNR and 0.030 SSIM over PerUser-HTTP, and

[4]Table 13 in Appendix D lists the exact fractions.

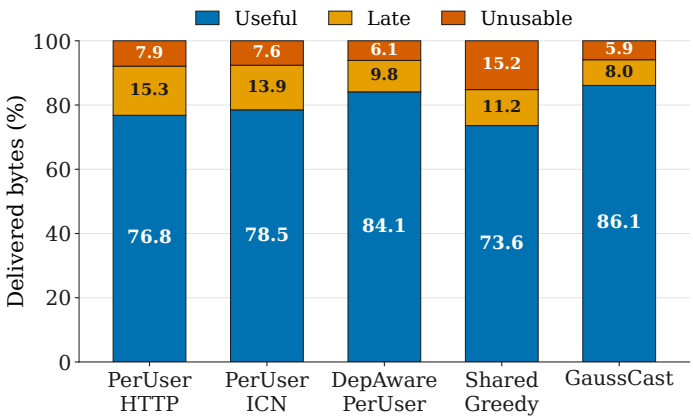


Fig. 7 Decomposition of delivered bytes into useful, late, and unusable fractions for each policy. Dependency-aware planning converts a larger fraction of delivered bytes into renderable content.

Table 9 Quality and fairness at matched playback times and bandwidth seeds.

Policy	PSNR ↑	SSIM ↑	LPIPS ↓	Jain ↑
PerUser-HTTP	28.05 ± 0.58	0.879 ± 0.015	0.182 ± 0.014	0.887 ± 0.021
DependencyAware-PerUser	28.74 ± 0.52	0.898 ± 0.013	0.162 ± 0.012	0.901 ± 0.020
SharedGreedy	28.31 ± 0.55	0.889 ± 0.014	0.171 ± 0.013	0.898 ± 0.020
GaussCast-Full	$\mathbf{29.02 \pm 0.49}$	$\mathbf{0.909 \pm 0.012}$	$\mathbf{0.153 \pm 0.011}$	$\mathbf{0.926 \pm 0.017}$

by 0.71 dB and 0.020 over SharedGreedy; the perceptual LPIPS distance improves in step (0.182 to 0.153 over PerUser-HTTP). Consistent with the 2×2 ablation, most of the quality gain comes from closure admission rather than sharing: DependencyAware-PerUser, which adds closure to a per-user substrate, already reaches 28.74 dB (0.69 dB over PerUser-HTTP and above SharedGreedy's 28.31 dB despite doing no cross-user sharing), and GaussCast adds a further 0.28 dB on top by combining closure with sharing. The fairness gain (Jain 0.887 to 0.926) indicates that the shared base does not advantage majority viewpoints at the expense of users whose views are less common: because the supplement stage uses fair rotation and a QoE-based weight, budget freed by shared prerequisites is redistributed to users with lower recent quality.

7.6 Mechanism Ablation

Table 10 isolates the contribution of planning, closure tests, cache retention, request aggregation, and the delivery mechanism. The ablation separates two distinct effects. Request aggregation and asymmetric cache retention mainly reduce *upstream bytes*: removing aggregation raises traffic from 0.74 to 0.86, and the cache-only variant still reaches 0.80 because retained lower layers serve late-arriving users even without same-cycle collapsing. The closure test mainly governs *unusable bytes*: variants without

Table 10 Mechanism ablation. Cl./Ca./Ag. mark whether the closure test, asymmetric cache, and request aggregation are enabled (✓) or not (×); "lim." marks limited aggregation. Upstream bytes are normalized to PerUser-HTTP; unusable bytes are a percentage of delivered bytes.

Variant	Cl.	Ca.	Ag.	Up. bytes	Late miss	Unus. bytes
PerUser-HTTP	×	std	×	1.00	0.184	7.9
PerUser-ICN	×	std	✓	0.93	0.166	7.6
DependencyAware-PerUser	✓	std	✓	0.89	0.108	6.1
SharedGreedy	×	✓	✓	0.82	0.147	15.2
GC-noClosure	×	✓	✓	0.79	0.161	14.1
GC-noAggr.	✓	✓	×	0.86	0.106	6.3
GC-cacheOnly	✓	✓	lim.	0.80	0.101	6.1
GC-Full	✓	✓	✓	**0.74**	**0.096**	**5.9**

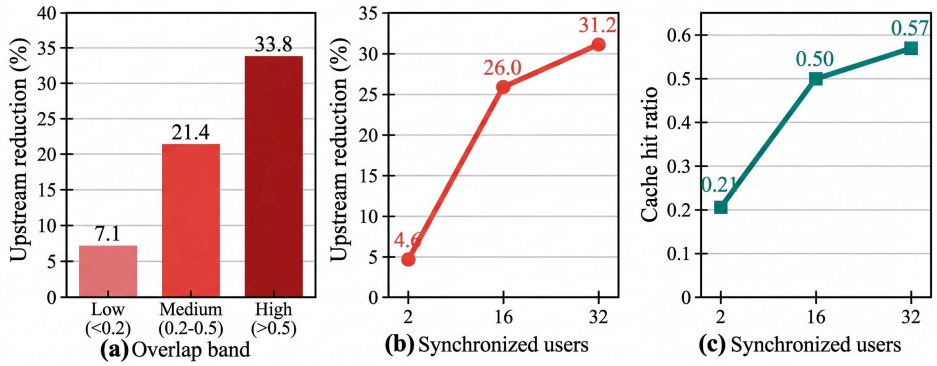


Fig. 8 Sensitivity to overlap and concurrency: (a) upstream-byte reduction over PerUser-HTTP by lower-layer overlap band; (b) upstream-byte reduction and (c) edge-cache hit ratio as the synchronized group scales from 2 to 32 users.

closure (SharedGreedy at 15.2%, GaussCast-noClosure at 14.1%) deliver many refinements before their prerequisites arrive, so those bytes cannot be rendered, whereas every closure-enabled variant stays near 6%. DependencyAware-PerUser makes this independent of sharing: it enables closure on a per-user substrate, so its unusable fraction falls to 6.1% even though it does no cross-user sharing and its upstream bytes stay high at 0.89. This confirms that byte savings and useful-delivery are produced by different mechanisms, and that naive sharing without dependency-aware admission trades duplicate traffic for wasted refinement traffic rather than eliminating waste.

7.7 Overlap, Concurrency, and Cold Start

Figure 8 reports performance as a function of overlap and user count. GaussCast should provide little gain when users explore unrelated regions and stronger gain when lower-layer overlap and concurrency grow.

Table 11 Pre-download break-even analysis.

Scene	Comp. full size	Full DL @10 Mbps	Full DL @20 Mbps	L0/L1 size	GaussCast TTFF
Room	38 MB	30.4 s	15.2 s	6.1 MB	0.39 s
Truck	142 MB	113.6 s	56.8 s	19.8 MB	0.56 s
Berlin	1.27 GB	1016 s	508 s	121 MB	0.74 s

The trend matches this expectation. We bin planning windows by measured lower-layer overlap (the Jaccard similarity of requested L0/L1 cells across concurrent users) into low (< 0.2), medium (0.2–0.5), and high (> 0.5) bands. Upstream-byte reduction over PerUser-HTTP grows from 7.1% in low-overlap windows to 21.4% at medium and 33.8% at high overlap, while in the low band GaussCast converges toward per-user delivery rather than degrading below it. Concurrency shows the same monotone pattern. Holding the trace mix fixed and scaling the synchronized group from 2 to 32 users, byte reduction rises from 4.6% to 31.2% and cache hit ratio from 0.21 to 0.57, passing through the 16-client default (26% reduction, 0.50 hit ratio) of Table 7. The cause is that more concurrent viewers raise the probability that a fetched lower layer is reused before eviction, consistent with the duplication factor climbing from 1.4 at two users to 6.4 at sixteen and 9.1 at thirty-two (Table 6).

Cold-start joins benefit specifically from cache retention. At the 5 s offset whose inter-user L0/L1 overlap is 0.27 (Table 6), a joining user finds 0.18 of its required L0/L1 closure already resident in the edge cache—lower than the overlap because some overlapping prerequisites have not yet been fetched or have aged out—which cuts its TTFF from 0.75 s to 0.58 s. Three TTFF figures appear in this paper and measure different conditions. The 0.55 s mean in Table 7 averages all joins across the three scenes and all windows in the 16-client run, mixing users who join into a partly warm shared cache with those who join cold; this value is close to the per-scene cold-empty values of Table 11 (0.39/0.56/0.74, mean $\approx$ 0.56 s) because the faster small scenes and the warm joins roughly offset the slower *Berlin* cold case. The 0.58 s above is one cold-start joiner at a 5 s offset whose shared cache is already partly warm, so it sits between the 0.75 s per-user join cost and the warm-join end of the distribution. The 0.74 s in Table 11 is the worst case: a cold join to the largest scene (*Berlin*) with an empty edge cache.

7.8 Pre-Download Boundary, Compression, and Dependency Sparsity

For each scene, we compare full compressed-scene download time with progressive TTFF and then scale block sizes to emulate stronger compression. This byte-scaling study isolates the effect of smaller blocks with the same demand and dependency structure; it is not a replacement for codec-specific evaluation. If full compressed download fits within the startup budget, pre-download is the simpler deployment choice.

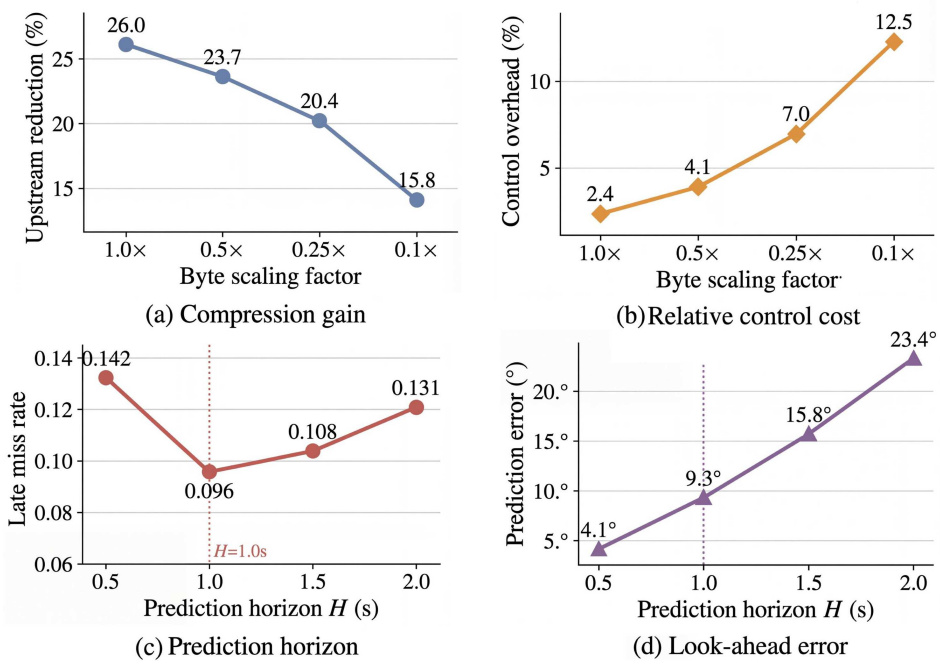


Fig. 9 Sensitivity of GaussCast to block size and prediction horizon: (a) upstream-byte reduction and (b) control overhead versus the byte-scaling factor; (c) late-miss rate and (d) prediction error versus the horizon H. Per-metric values appear in Tables 14 and 16.

The break-even analysis shows where progressive delivery is justified. For *Room* (38 MB compressed), full download completes in 15–30 s; this is large relative to an interactive startup budget but tractable, so pre-download is a defensible choice. For *Berlin* (1.27 GB even after compression), full download takes 8–17 minutes, whereas GaussCast reaches first frame in 0.74 s by fetching only the prerequisite closure of the cells visible at the join pose—a few megabytes—rather than the scene's full 121 MB base-layer prefix or the 1.27 GB complete model. The remaining base-layer blocks stream in progressively as the user moves and are increasingly served from the shared edge cache, so the 121 MB prefix in Table 11 bounds the eventual base coverage, not the bytes on the critical path to the first frame. The compression-sensitivity study (Figure 9a,b) confirms that the benefit shrinks but does not vanish as blocks get smaller: at 0.1× scale (emulating an aggressive future codec) upstream reduction falls from 26.0% to 15.8%, and control overhead rises from 2.4% to 12.5% of delivered bytes because fixed-size manifests and plan messages become a larger fraction of much smaller data blocks. Late miss falls in the same direction (0.096 to 0.063) because smaller blocks drain faster and fit within their deadlines more easily, so the byte-saving and deadline effects move oppositely as compression strengthens. The cross-user overlap that GaussCast exploits is unchanged by byte scaling, so the gain is tied to overlap and scene scale rather than to any particular block size; once blocks become small enough that control overhead approaches the upstream saving, pre-download or coarser sharing becomes the better choice.

Dependency sparsity.

To test whether the results depend on the conservative full-prefix dependency, we also vary the dependency density itself as a stress test. This does not assume that every layered codec exposes these exact densities; it shows how the planner behaves as the amount of prerequisite data changes.

As dependency density increases (Table 15, Appendix D), more lower-layer data is shared across users, so upstream bytes drop from 0.88 at 25% density to 0.74 at the full-prefix setting—the regime GaussCast is designed for. Late miss rises modestly (0.071 to 0.096) because denser prerequisites lengthen each closure and leave less slack before refinement deadlines, yet it stays well below the per-user baseline at every density. Useful-byte share declines only slightly (88.4% to 86.1%) and quality improves with density because more complete prerequisites enable higher-layer refinements; even the sparsest setting already exceeds SharedGreedy's PSNR, since closure-aware admission helps quality independently of how much prerequisite data each block carries. The planner thus degrades gracefully toward per-user behavior under sparse dependencies rather than failing, and its largest gains coincide with the conservative full-prefix model that real layered 3DGS pipelines expose. This study and the byte-scaling study above agree: the gain shrinks smoothly as the shared prerequisite mass falls, whether because blocks are smaller or because fewer of them are required.

7.9 Prediction Horizon Sensitivity

Every score in GaussCast —the demand score $p_u(r)$, the sharing signal $a_x(r)$, and the useful deadline $t_u^{\mathrm{ddl}}(r)$—is derived from short-horizon pose prediction, so a natural concern is how strongly the gains depend on prediction quality. We vary the prediction horizon H and report the realized prediction error alongside delivery quality. Prediction error is measured as the median angular deviation between the predicted and ground-truth forward direction at the end of the horizon, computed directly from the EyeNavGS traces with the same constant-velocity-plus-gaze predictor used throughout. Because a longer horizon exposes more drift, H also acts as a prediction-error knob: it lengthens the prefetch lead time while simultaneously degrading prediction accuracy.

Figure 9c,d shows that the horizon trades prefetch lead time against prediction accuracy, and that late miss is U-shaped with a minimum at the 1.0 s default. At $H = 0.5$ s the predictor is most accurate (4.1° error), but the proxy has too little lead time to land each prerequisite closure before its deadline, so late miss rises to 0.142 even though few bytes are wasted. At $H = 2.0$ s the lead time is ample, but the 23.4° prediction error makes more speculative prefetches miss the user's actual view, so upstream bytes grow to 0.85 and the unusable fraction nearly doubles to 10.2%. The 1.0 s setting balances the two effects and is the configuration used in all other experiments. A further point is that prediction error inflates waste only gradually. Even at the longest, least accurate horizon, the unusable fraction stays at 10.2%, below the 14.1% that GaussCast-noClosure already incurs at the well-predicted 1.0 s horizon, because the closure and deadline tests refuse to admit blocks that cannot arrive in time for the predicted view, so a misprediction causes a block to be skipped rather than fetched and discarded. This validates the design rationale of Section 5: explicit

Table 12 Verification and control overhead. The PK-verif./s and CPU/block columns use different denominators (per-block public-key operations versus CPU amortized over all blocks); latency columns apply to the digest-linked modes only.

Mode	PK verif./s	CPU/ block	Extra mf. RTTs	Retr. lat.	Ctrl/data bytes
Per-block signature	401	2.49 ms	—	—	—
Digest-linked manifest	52	0.34 ms	0.63	86 ms	1.9%
Digest-linked + hints	31	0.21 ms	0.36	61 ms	2.4%

closure, budget, and deadline checks bound the cost of prediction failure instead of relying on the predictor being correct.

7.10 Verification and Control Overhead

Verification cost has two parts: cryptographic CPU load and retrieval-path latency. Table 12 reports both, comparing per-block signing against digest-linked manifests with and without plan hints.

Per-block public-key signing is the dominant cryptographic cost: it forces a public-key verification on every block, reaching 401 operations per second under concurrent demand and 2.49 ms CPU per block. A signed manifest plus digest-linked chunks moves almost all checks to hashes—only manifests need public-key verification—cutting public-key operations to 52/s, and to 31/s once plan hints batch manifest subtrees, while per-block CPU falls to 0.21 ms. Plan hints also let the proxy prefetch the manifest subtrees the planner will need, cutting extra manifest round trips from 0.63 to 0.36 per block and retrieval latency from 86 ms to 61 ms. Control traffic stays at 2.4% of data bytes, the planning step adds 4.4 ms on average (6.8 ms at the 95th percentile) per cycle, and no invalid block was accepted in any mode, so coordination cost remains small relative to the byte and quality gains.

7.11 Online Adaptation and Robustness

Two further studies check whether the headline gains depend on the optional controller or the controlled emulation.[5] First, on the learned controller: as already noted in Section 5.7, the deterministic planner under fixed parameters reaches upstream bytes ≈ 0.77 and TTFF ≈ 0.58 s on its own, and the PPO controller adds the remainder, staying ahead of rule-based and grid-search adaptation but within the band the deterministic planner defines (Table 17). Second, the trend holds outside the emulator: a cloud/WAN pilot at 41 ms RTT raises upstream bytes only from 0.74 to 0.77 and TTFF from 0.55 s to 0.68 s, because cache reuse and aggregation are RTT-independent once prerequisites are resident, and an update-churn microbenchmark shows versioned manifests localize invalidation: updating 10% of chunks per minute invalidates only 10.8% of cache (Tables 18–19). The pilot is a single-scene sanity check rather than a full deployment study, but together these indicate that closure-aware planning, not

[5] We summarize both studies here and report the full tables in Appendix E.

the learned component or the emulated topology, drives the result, while frame-rate dynamic content remains future work.

8 Discussion and Limitations

When* GaussCast *helps and when it does not.

GaussCast benefits only the regime defined in Section 2, and degrades by design outside it: the overlap, sparse-dependency, and strong-compression studies all show the planner shifting budget to per-user supplements as the reuse opportunity shrinks, and for scenes whose compressed model already fits the startup budget, pre-download remains the simpler choice. The system adapts to the measured workload instead of forcing counterproductive sharing.

Updates and cache validity.

The prototype assumes stable scene versions within planning windows. When the publisher updates the scene, new objects are released under a new version namespace and delta manifests. Unchanged subtrees retain their digests (because block names include content-addressed digests, identical content yields identical names) and remain cached; updated blocks receive new digests and are fetched as distinct objects. Clients do not mix blocks across versions unless the manifest declares unchanged compatible subtrees. Highly dynamic 3DGS video—where scene content changes at frame rates—would require temporal dependency edges and stricter expiry rules, which is future work.

Application scenarios.

GaussCast targets shared virtual tours (museum exhibits, real-estate walkthroughs), XR classrooms (instructor-led exploration), collaborative design reviews (architecture, engineering), and digital-twin exploration (facility management, urban planning). These scenarios share characteristics: multiple concurrent users, large-scale scenes, overlapping viewing regions, and cold-cache joins. For single-user streaming or scenarios where users explore entirely disjoint regions, existing per-user adaptive streaming systems remain appropriate.

Security and privacy.

The publisher is trusted for content authenticity. Proxies and forwarders are trusted for performance but not content integrity. Digest-linked verification prevents proxies from authenticating unauthenticated data, while versioned names and freshness checks reduce stale replay. Clients report compact session summaries revealing coarse viewing intent to the edge proxy. Upper-level proxies use aggregate block-demand summaries instead of raw pose or gaze traces. Stronger privacy can use coarse visible-cell sketches or block-demand filters, trading accuracy for privacy. The prototype does not address traffic-analysis attacks on timing or object-name sequences.

Trace construction and deployment model.

The evaluation uses real 6DoF traces but forms concurrent sessions by grouping trace windows with controlled start-time offsets because source traces were recorded individually. This controlled setup makes overlap and cold-start timing measurable but does not fully model social interaction or correlated intent in live events. Our evaluation spans the three spatial regimes of Table 1 (bounded indoor, object-centric outdoor, and gigabyte-scale multi-room indoor), which is the range over which the mechanism's behavior is determined: GaussCast depends only on per-cell lower-layer overlap, dependency closures, and the shared upstream bottleneck, none of which are specific to a particular scene category. A city-scale model is, in principle, a larger instance of the same regime, with more cells and deeper hierarchies under the identical block, dependency, and overlap abstractions, so we expect the same mechanisms to apply. The magnitude of the gains may differ if larger working sets or deeper hierarchies change the cache-hit rate or the planner's candidate-set size, which are implementation limits rather than changes to the dependency-closure or shared-retrieval abstractions themselves. We do not evaluate a city-scale deployment directly: at that scale the active candidate set, cache working set, and cross-user overlap statistics may behave differently from the gigabyte-scale *Berlin* scene we measure, and confirming the trends there remains future work. The current objective focuses on shared upstream paths. Deployments where last-hop wireless airtime dominates should add downstream airtime to the budget model; the dependency-closure test remains unchanged.

The one-cycle planner is greedy. Because block values interact through shared closures and upstream costs, GaussCast does not claim constant-factor approximation guarantees. Proposition 1 holds only when completion-time estimates are conservative. Short-horizon prediction can fail under abrupt motion; the design uses explicit closure, budget, cache, and deadline checks, and Section 7.9 shows that these checks bound the cost of prediction error rather than letting it inflate wasted traffic—the unusable fraction grows only gradually as the horizon lengthens and prediction degrades. Evaluation reports late misses and planner latency against strong baselines instead of relying on single weighted scores.

Retrieval mechanism differences.

Section 6 notes that same-cycle aggregation depends on deployment: PIT-based aggregation helps only while Interests for the same segment are pending. HTTP/CDN or relay deployments need request collapsing or relay logic to match it. Users arriving later are served by cache retention rather than aggregation, so cache policy carries more weight in sessions with low arrival synchrony.

9 Conclusion

GaussCast addresses a delivery challenge that persists after layered 3DGS representation and compression: when multiple users render different views of large-scale scenes, they often depend on overlapping lower-layer blocks. By structuring scenes as named, cacheable, versioned retrieval blocks with explicit dependency metadata, an edge proxy can fetch reusable prerequisites once and add user-specific supplements.

The closure-aware planner uses cache state, deadlines, and dependency information to avoid scheduling refinements before required lower layers can arrive—a coordination problem that generic caches, request aggregation, and relays reduce but do not solve.

The resulting design is most appropriate for shared, progressive 3DGS viewing with measurable lower-layer overlap and cold or changing cache state. For small scenes where full compressed download fits startup budgets, pre-download remains simpler. For large-scale environments under bandwidth or latency constraints with concurrent user access, GaussCast reduces shared edge-to-origin upstream traffic by 26%, lowers time-to-first-frame by 27%, and improves quality and fairness, with gains that grow with cross-user overlap in the evaluated workload.

Statements and Declarations

Funding. The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Conflict of interest. The authors have no relevant financial or non-financial interests to disclose.

Data availability. The 6DoF navigation traces used in the evaluation are derived from the publicly available EyeNavGS dataset, and the source 3DGS scenes are obtained from the public Mip-NeRF 360, Tanks and Temples, Deep Blending, and Zip-NeRF collections. The layered-block manifests, demand logs, and emulation configurations generated for this study, together with the scripts that regenerate them from the public EyeNavGS traces, are available in the public code repository listed below.

Code availability. A prototype implementation, including the dependency-aware planner, layered-block manifest generator, trace-driven delivery simulator, Mini-NDN/NFD and Mininet emulation harnesses, and the plotting scripts that reproduce the reported figures and tables, is publicly available at https://github.com/ulen2000/gausscast-sim.

A Trace Construction and the Synthetic Guided Model

The released EyeNavGS traces record free exploration: each of the 46 participants navigates a scene individually, so the dataset contains no instructor-led or class-like sessions. We therefore report all free-exploration and cold-start rows of Table 6 directly from the measured traces, and model the guided/class-like row synthetically rather than claiming a measurement we do not have.

The synthetic guided model reuses real motion statistics. For each scene we select one participant's free-exploration path as a guide and generate followers by perturbing the guide pose with additive Gaussian jitter whose standard deviations (σ_{pos} in position, σ_{ang} in look direction) are calibrated to the inter-participant deviation measured in the real traces. This produces correlated-but-not-identical viewing, the regime an instructor-led session approximates, while keeping per-frame motion drawn from real data. The model raises lower-layer overlap from the free-exploration value 0.23 to

0.43, which we report as a model output, not a measurement. We mark this row with a dagger throughout and exclude it from any headline claim; the end-to-end results in Section 7 are computed over measured free-exploration windows.

B Optional Controller: Action, State, and Reward

This appendix gives the exact definitions for the optional parameter controller of Section 5.7. At cycle t the controller outputs the high-level action

$$\kappa_t = [r_t^{\mathrm{sh}}, \eta_2, \eta_3, \eta_4, \tau_t^{\mathrm{keep}}, \tau_t^{\mathrm{sup}}], \tag{26}$$

where r_t^{sh} is the shared-budget ratio, η_2–η_4 tune dependency, cost, and congestion terms, τ_t^{keep} controls cache retention, and τ_t^{sup} controls the supplement threshold. The state contains only aggregate signals:

$$\xi_t = [|U_x|, \overline{\mathrm{ov}}_x, \mathrm{cacheOcc}, \mathrm{queueBacklog}, \overline{\mathrm{RTT}}, \overline{\mathrm{tput}}, \mathrm{missRatio}, \mathrm{lateRatio}, \mathrm{Jain}_t]. \tag{27}$$

The reward balances delivered utility, upstream bytes, late misses, fairness, and cache reuse:

$$R_t^{\mathrm{ctrl}} = \Delta\mathrm{QoE}_t - \lambda_1 \mathrm{UpBytes}_t - \lambda_2 \mathrm{LateMiss}_t - \lambda_3(1 - \mathrm{Jain}_t) + \lambda_4 \mathrm{HitGain}_t. \tag{28}$$

C Active Candidate Set and Safety Checks

The proxy does not score every block in the full scene during every cycle. It builds the active candidate set $\mathcal{A}_x$ using frustum filtering, recent pose history, layer limits, predicted motion, and cache state. Blocks outside the prediction horizon or below a minimum value threshold are pruned. This keeps one-cycle planning latency bounded by the number of visible or near-visible blocks rather than the full scene size.

All optional optimization modules stay outside the trust boundary. The controller may change high-level parameters, cache retention changes only the eviction priority of valid blocks, and plan-carried verification hints only shorten manifest traversal. None of these modules can authenticate data, mix incompatible scene versions, or admit a block that fails the dependency-closure and deadline tests.

D Detailed Sensitivity Tables

Figure 9 in the main text summarizes the compression and prediction-horizon sensitivity studies; the dependency-sparsity study is discussed inline in Section 7. Table 13 lists the exact delivered-byte fractions plotted in Figure 7, and Tables 14–16 report the full per-metric values for the sensitivity studies.

E Online Adaptation and Robustness Checks

This appendix reports the learned-controller comparison and two robustness checks that support, but are not central to, the main results: the deterministic closure-aware

Table 13 Delivered-byte decomposition (% of delivered bytes), corresponding to Figure 7.

Policy	Useful ↑	Late ↓	Unusable ↓
PerUser-HTTP	76.8	15.3	7.9
PerUser-ICN	78.5	13.9	7.6
DependencyAware-PerUser	84.1	9.8	6.1
SharedGreedy	73.6	11.2	15.2
GaussCast-Full	**86.1**	**8.0**	**5.9**

Table 14 Compression sensitivity with fixed access traces and scaled block sizes.

Byte scale	Upstream reduction	TTFF	Late miss	Control overhead
1.0×	26.0%	0.55 s	0.096	2.4%
0.5×	23.7%	0.38 s	0.082	4.1%
0.25×	20.4%	0.29 s	0.071	7.0%
0.1×	15.8%	0.24 s	0.063	12.5%

Table 15 Sensitivity to dependency sparsity.

Dependency density	Upstream bytes	Useful bytes	Late miss	PSNR/SSIM
25%	0.88	88.4	0.071	28.46 / 0.894
50%	0.83	87.6	0.079	28.68 / 0.901
75%	0.78	86.9	0.088	28.87 / 0.906
100% full-prefix	0.74	86.1	0.096	29.02 / 0.909

planner already delivers most of the reported gain, and these studies indicate that the optional controller and the controlled emulation do not drive the headline numbers.

Online adaptation.

We compare the PPO controller with simple and strong alternatives. Static parameters keep one fixed configuration. RuleAdaptive changes the shared-budget ratio using measured overlap, cache occupancy, and late-miss rate. OnlineGrid tests a small set of parameter choices on recent windows. OracleTuned uses hindsight and serves as an upper bound. This comparison separates online parameter tuning from the deterministic planner.

All adaptive controllers improve on static parameters, confirming that the shared-budget ratio benefits from adjustment as bandwidth and overlap drift. PPO raises utility by 7.6% over static while cutting late miss to 0.835× and upstream bytes

Table 16 Sensitivity to prediction horizon. Prediction error is the median forward-direction error at the end of the horizon. The default 1.0 s row reproduces the main results.

Horizon H	Pred. error	Upstream bytes	Useful bytes	Late miss	Unusable bytes
0.5 s	4.1°	0.71	82.0	0.142	6.4
1.0 s	9.3°	0.74	**86.1**	**0.096**	5.9
1.5 s	15.8°	0.79	84.7	0.108	7.7
2.0 s	23.4°	0.85	81.3	0.131	10.2

Table 17 High-level parameter adaptation under dynamic bandwidth and overlap. Utility, upstream bytes, and late miss are multipliers relative to the static-parameter planner; Jain and inference time are absolute values.

Controller	Utility	Upstream bytes	Late miss	Jain	Inference time
Static parameters	1.00	1.00	1.00	0.907	0
RuleAdaptive	1.035	0.985	0.92	0.915	0.15 ms
OnlineGrid	1.050	0.975	0.88	0.919	1.6 ms
PPO	**1.076**	**0.962**	**0.835**	**0.926**	0.7 ms
OracleTuned	1.094	0.945	0.80	0.931	offline

by 3.8%, reaching 81% of the OracleTuned utility gain at 0.7 ms inference. Its margin over RuleAdaptive (+4.0%) and OnlineGrid (+2.5%) is real but moderate: the deterministic closure-aware planner already enforces correctness, so the controller only tunes high-level parameters and cannot fix scheduling mistakes. RuleAdaptive captures roughly half of PPO's gain at negligible cost and is a reasonable choice when an RL pipeline is undesirable; PPO is most valuable when bandwidth and overlap shift faster than hand-tuned rules track.

Real network and update churn.

The main results use controlled emulation. We add two robustness checks: a real-network pilot that runs the same implementation under live RTT and throughput, and an update-churn microbenchmark for versioned manifests.

Table 18 Real-network pilot.

Deployment	RTT	Throughput	Upstream bytes	TTFF
Emulated (main)	18 ms	12–40 Mbps	0.74	0.55 s
Cloud/WAN pilot	41 ms	9–46 Mbps	0.77	0.68 s

The cloud/WAN pilot (Table 18) runs the same implementation across a public cloud region with 41 ms RTT and uncontrolled throughput, averaged over 10 sessions

Table 19 Update-churn microbenchmark.

Updated chunks/min	Delta size	Invalidated cache	Hit ratio	TTFF
1%	0.4 MB	1.1%	0.49	0.56 s
5%	2.1 MB	5.3%	0.46	0.59 s
10%	4.3 MB	10.8%	0.42	0.63 s

on the *Truck* scene. Upstream bytes rise only slightly (0.74 to 0.77) and TTFF from 0.55 s to 0.68 s, so the observed trend persists in this pilot under real wide-area RTT and throughput rather than only under the emulated topology; longer RTT mainly affects startup rather than steady-state byte savings because cache reuse and aggregation are RTT-independent once prerequisites are resident. This is a single-scene, ten-session sanity check rather than a complete deployment study: it does not vary scene, client count, or network conditions, and a full multi-scene, multi-region characterization remains future work. The churn microbenchmark (Table 19) confirms that versioned manifests localize invalidation: updating 10% of chunks per minute invalidates only 10.8% of cache and lowers the hit ratio from 0.49 to 0.42, since unchanged subtrees retain their digests and stay cached. TTFF grows modestly (0.56 s to 0.63 s) because clients refetch only changed lower layers. This supports the design claim that slowly changing scenes do not force whole-scene re-download, while genuinely high-rate dynamic content—changing at frame rates—remains future work.

References

[1] Kerbl, B., Kopanas, G., Leimkühler, T., Drettakis, G.: 3d gaussian splatting for real-time radiance field rendering. ACM Transactions on Graphics **42**(4) (2023)

[2] Lu, T., Yu, M., Xu, L., Xiangli, Y., Wang, L., Lin, D., Dai, B.: Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 20654–20664 (2024)

[3] Ren, K., Jiang, L., Lu, T., Yu, M., Xu, L., Ni, Z., Dai, B.: Octree-gs: Towards consistent real-time rendering with lod-structured 3d gaussians. CoRR **abs/2403.17898** (2024)

[4] Liu, Y., Guan, H., Luo, C., Fan, L., Wang, N., Peng, J., Zhang, Z.: Citygaussian: Real-time high-quality large-scale scene rendering with gaussians. In: European Conference on Computer Vision (ECCV), pp. 265–282 (2024). https://doi.org/10.1007/978-3-031-72640-8_15

[5] Kerbl, B., Meuleman, A., Kopanas, G., Wimmer, M., Lanvin, A., Drettakis, G.: A hierarchical 3d gaussian representation for real-time rendering of very large datasets. ACM Transactions on Graphics **43**(4) (2024)

[6] Fan, Z., Wang, K., Wen, K., Zhu, Z., Xu, D., Wang, Z.: Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps. In: Advances in Neural Information Processing Systems, vol. 37 (2024)

[7] Niedermayr, S., Stumpfegger, J., Westermann, R.: Compressed 3d gaussian splatting for accelerated novel view synthesis. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 10349–10358 (2024). https://doi.org/10.1109/CVPR52733.2024.00985

[8] Chen, Y., Wu, Q., Lin, W., Harandi, M., Cai, J.: Hac: Hash-grid assisted context for 3d gaussian splatting compression. In: European Conference on Computer Vision (ECCV), pp. 422–438 (2024). https://doi.org/10.1007/978-3-031-72667-5_24

[9] Girish, S., Gupta, K., Shrivastava, A.: Eagles: Efficient accelerated 3d gaussians with lightweight encodings. In: European Conference on Computer Vision (ECCV), pp. 54–71 (2024). https://doi.org/10.1007/978-3-031-73036-8_4

[10] Chen, Y., Wu, Q., Lin, W., Harandi, M., Cai, J.: Hac++: Towards 100x compression of 3d gaussian splatting. IEEE Transactions on Pattern Analysis and Machine Intelligence (2025)

[11] Bagdasarian, M.T., Knoll, P., Breuss, M., Weinmann, M.: A survey on 3d gaussian splatting compression methods. Computer Graphics Forum (2025) https://doi.org/10.1111/cgf.70078

[12] Tsai, Y.-Z., Zhang, X., Li, Z., Chen, J.: L3gs: Layered 3d gaussian splats for efficient 3d scene delivery. CoRR **abs/2504.05517** (2025)

[13] Shi, Y., Morin, G., Gasparini, S., Ooi, W.T.: Lapisgs: Layered progressive 3d gaussian splatting for adaptive streaming. CoRR **abs/2408.14823** (2024)

[14] Sun, Y.-C., Shi, Y., Lee, C.-T., Zhu, M., Ooi, W.T., Liu, Y., Huang, C.-Y., Hsu, C.-H.: Lts: A dash streaming system for dynamic multi-layer 3d gaussian splatting scenes. In: Proceedings of the 16th ACM Multimedia Systems Conference, pp. 136–147 (2025). https://doi.org/10.1145/3712676.3714445

[15] Chen, Y., Lee, G.H.: Dogs: Distributed-oriented gaussian splatting for large-scale 3d reconstruction via gaussian consensus. In: Advances in Neural Information Processing Systems, vol. 37 (2024)

[16] Liu, Y., Luo, C., Mao, Z., Peng, J., Zhang, Z.: CityGaussianV2: Efficient and Geometrically Accurate Reconstruction for Large-Scale Scenes (2024)

[17] Chen, Y., Li, M., Wu, Q., Lin, W., Harandi, M., Cai, J.: Pcgs: Progressive compression of 3d gaussian splatting. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 40, pp. 3111–3119 (2026)

[18] Tang, Z., Zhang, Q., Zhang, L., Hou, J., Yang, Y., Wang, X.: Gsstream: 3d gaussian splatting based volumetric scene streaming system. CoRR **abs/2603.09718** (2026)

[19] Gong, H., Li, Q., Liu, Z., Zhou, H., Zhou, P., Li, Z., Li, J.: Adaptive 3D Gaussian Splatting Video Streaming (2025)

[20] Yin, D., Jin, Y., Shi, J., Ding, I., Zhang, M., Wang, F., Huang, Z., Zhang, C., Liu, J., Dong, F.: CAGS: Color-Adaptive Volumetric Video Streaming with Dynamic 3D Gaussian Splatting (2026)

[21] Hosseini, M., Swaminathan, V.: Adaptive 360 vr video streaming based on mpeg-dash srd. In: IEEE International Symposium on Multimedia (2016)

[22] Ozcinar, C., De Abreu, A., Smolic, A.: Viewport-aware adaptive 360 degree video streaming using tiles for virtual reality. In: IEEE International Conference on Image Processing (2017)

[23] Mahmoud, M., Rizou, S., Panayides, A.S., Kantartzis, N.V., Karagiannidis, G.K., Lazaridis, P.I., Zaharis, Z.D.: A survey on optimizing mobile delivery of 360 degree videos: Edge caching and multicasting. IEEE Access (2023)

[24] Maniotis, P., Thomos, N.: Viewport-aware deep reinforcement learning approach for 360-degree video caching. arXiv preprint arXiv:2003.08473 (2020)

[25] Hooft, J., Wauters, T., De Turck, F., Timmerer, C., Hellwagner, H.: Towards 6dof http adaptive streaming through point cloud compression. In: Proceedings of the 27th ACM International Conference on Multimedia (2019)

[26] Liu, Y., Qian, F., Mao, Z.M., Han, B.: Vues: Practical mobile volumetric video streaming through multiview transcoding. In: Proceedings of the 28th Annual International Conference on Mobile Computing and Networking (2022)

[27] Zhang, D., Zhou, P., Pathak, P.H.: M5: Facilitating multi-user volumetric content delivery with multi-lobe multicast over mmwave. In: Proceedings of the ACM International Conference on Mobile Computing and Networking (2022)

[28] Liu, Y., Zhou, P., Zhang, Z., Zhang, A., Han, B., Li, Z., Qian, F.: Muv2: Scaling up multi-user mobile volumetric video streaming via content hybridization and sharing. In: Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom), pp. 312–326 (2024). https://doi.org/10.1145/3636534.3649364

[29] Jin, Y., Hu, K., Liu, J., Wang, F., Liu, X.: From capture to display: A survey on volumetric video. arXiv preprint arXiv:2309.05658 (2024)

[30] Zhang, L., Afanasyev, A., Burke, J., Jacobson, V., Claffy, K.C., Crowley, P.,

Papadopoulos, C., Wang, L., Zhang, B.: Named data networking. ACM SIGCOMM Computer Communication Review **44**(3), 66–73 (2014) https://doi.org/10.1145/2656877.2656887

[31] Afanasyev, A., Burke, J., Refaei, T., Wang, L., Zhang, B., Zhang, L.: A brief introduction to named data networking. IEEE Military Communications Conference (2018)

[32] Chai, R., Zhu, Y., Li, X., Li, J., Meng, Z., Kutscher, D.: Inds: Incremental named data streaming for real-time point cloud video. In: Proceedings of the 33rd ACM International Conference on Multimedia, pp. 12102–12110 (2025)

[33] Mosko, M., Solis, I., Wood, C.: Information-Centric Networking (ICN): Content-Centric Networking (CCNx) and Named Data Networking (NDN) Terminology. RFC 8793 (2020). https://doi.org/10.17487/RFC8793

[34] Dabirmoghaddam, A., Dehghan, M., Garcia-Luna-Aceves, J.J.: Characterizing interest aggregation in content-centric networks. In: 2016 IFIP Networking Conference (IFIP Networking) and Workshops, pp. 449–457 (2016). https://doi.org/10.1109/IFIPNetworking.2016.7497205

[35] Fielding, R., Nottingham, M., Reschke, J.: RFC 9111: HTTP Caching. RFC Editor (2022)

[36] Tschudin, C., Wood, C.A., Mosko, M., Oran, D.R.: File-like icn collections (flic). Internet-Draft draft-irtf-icnrg-flic-07, Internet Engineering Task Force (mar 2025). Work in Progress. https://www.ietf.org/archive/id/draft-irtf-icnrg-flic-07.html

[37] Moiseenko, I.: Fetching content in named data networking with embedded manifests. Technical Report NDN-0025, Named Data Networking Project (sep 2014). Revision 1. https://named-data.net/publications/techreports/ndn-tr-25-manifest-embedding/

[38] Shin, Y.-Y., Park, S.H., Thai, Q.T., Byun, S.H.: Canary: A scalable content integrity verifying protocol for icn. In: Proceedings of the 6th ACM Conference on Information-Centric Networking, pp. 165–166 (2019)

[39] Jain, R., Chiu, D.-M., Hawe, W.R.: A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report DEC-TR-301, Digital Equipment Corporation (sep 1984)

[40] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. CoRR **abs/1707.06347** (2017)

[41] Rahman, A., Trossen, D., Kutscher, D., Ravindran, R.: Deployment Considerations for Information-Centric Networking (ICN). RFC 8763 (2020). https:

//doi.org/10.17487/RFC8763 . https://www.rfc-editor.org/info/rfc8763

[42] Bishop, M.: RFC 9114: HTTP/3. RFC Editor (2022)

[43] Mini-NDN Team: Mini-NDN: A Mininet-based NDN Emulator. https://mini-ndn.readthedocs.io/. Online; accessed 2026-03-31 (2025)

[44] NFD Team: Nfd developer's guide. Technical Report NDN-0021, Revision 11, Named Data Networking Project (aug 2021). https://named-data.net/publications/techreports/nfd-developer-guide/

[45] Lantz, B., Heller, B., McKeown, N.: A network in a laptop: Rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets), pp. 19–1196 (2010). https://doi.org/10.1145/1868447.1868466

[46] Ding, Z., Lee, C.-T., Zhu, M., Guan, T., Sun, Y.-C., Hsu, C.-H., Liu, Y.: Eyenavgs: A 6-dof navigation dataset and record-n-replay software for real-world 3dgs scenes in vr. arXiv preprint arXiv:2506.02380 (2025)

[47] Barron, J.T., Mildenhall, B., Verbin, D., Srinivasan, P.P., Hedman, P.: Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 5460–5469 (2022)

[48] Knapitsch, A., Park, J., Zhou, Q.-Y., Koltun, V.: Tanks and temples: Benchmarking large-scale scene reconstruction. ACM Transactions on Graphics **36**(4) (2017)

[49] Hedman, P., Philip, J., Price, T., Frahm, J.-M., Drettakis, G., Brostow, G.: Deep blending for free-viewpoint image-based rendering. ACM Transactions on Graphics (ToG) **37**(6), 1–15 (2018)

[50] Barron, J.T., Mildenhall, B., Verbin, D., Srinivasan, P.P., Hedman, P.: Zip-nerf: Anti-aliased grid-based neural radiance fields. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 19697–19705 (2023)

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- suppmultiuserplayback.mp4