Grace: Balancing Quality and Tail Delay in Real-Time Video via Data-Scalable Autoencoders

Yihua Cheng\textsuperscript{1}, Anton Arapin\textsuperscript{1}, Ziyi Zhang\textsuperscript{1}, Qizheng Zhang\textsuperscript{2}, Hanchen Li\textsuperscript{1}, Nick Feamster\textsuperscript{1}, Junchen Jiang\textsuperscript{1}
\textsuperscript{1}The University of Chicago, \textsuperscript{2}Stanford University

Abstract

Across many real-time video applications, we see a growing need (especially in long delays and dynamic bandwidth) to allow clients to decode each frame once any (non-empty) subset of its packets is received and improve quality with each new packet. We call it data-scalable delivery. Unfortunately, existing techniques (e.g., FEC, RS and Fountain Codes) fail short—they require either delivery of a minimum number of packets to decode frames, and/or pad video data with redundancy in anticipation of packet losses, which hurts video quality if no packets get lost. This work explores a new approach, inspired by recent advances of neural-network autoencoders, which make data-scalable delivery possible. We present Grace, a concrete data-scalable real-time video system. With the same video encoding, Grace’s quality is slightly lower than traditional codec without redundancy when no packet is lost, but with each missed packet, its quality degrades much more gracefully than existing solutions, allowing clients to flexibly trade between frame delay and video quality.

Grace makes following contributions. It trains new custom autoencoders to balance compression efficiency and resilience against a wide range of packet losses; and it uses a new transmission scheme to deliver autoencoder-coded frames as individually decodable packets. We test Grace (and traditional loss-resilient schemes and codecs) on real network traces and videos, and show that while Grace’s compression efficiency is slightly worse than heavily engineered video codecs, it significantly reduces tail video frame delay (by 2\times at the 95th percentile) with the marginally lowered video quality.

1 Introduction

With applications ranging from video conferencing to emerging IoT services and cloud-based gaming [2, 4, 12–15], real-time videos in Chrome over WebRTC (one of the most popular real-time video frameworks) grew 100x during 2020 [20], and one global industry survey found that services based on real-time videos grew from less than 5\% to over 28\% between 2020 and 2021 [10, 11].

With this increasing demand for real-time videos, it is crucial to deliver the video frames smoothly and with decent quality under a wide range of network conditions\textsuperscript{1}, including

\textsuperscript{1}This problem is so profound and urgent that serious efforts are finally being made by ISPs to offer active queue management for real-time traffic [3], marking a shift in focus from bandwidth to tail delay [17], but wide

---

Figure 1: Comparing different loss-resilient schemes when the actual encoded bitrate (not target bitrate) is set at 3.6 Mbps. (FEC bits are included in the encoded bitrate.) Quality of Grace’s autoencoder degrades gracefully with a higher packet loss rate. More results can be found in §5.
Grace, a new real-time video system based on the emergent neural autoencoders. Figure 1 shows an example of Grace’s loss resilience: if no packet is lost, a frame can be decoded at a similar quality to classic codecs (e.g., H.264), and with more packet losses, Grace’s quality degrades more gracefully than various baselines (explained in §5).

At its core, Grace’s new autoencoder offers a new abstraction, called data-scalable delivery: the same encoding of a frame can be decodable once any non-empty subset of its packets are received (as opposed to a minimum set of packets required by FEC, RS or fountain codes) and achieve higher quality with each new packet received. Therefore, a sender does not need to predict packet losses; instead, the receiver can flexibly choose when to decode a frame once any packets have arrived while enjoying decent video quality.

Grace customizes autoencoders because of their empirical smooth relationships between video quality and the perturbations on the coded data (e.g., packet losses), which naturally arise from autoencoders’ neural network structure (elaborated in §3.3). However, off-the-shelf autoencoders do not yet have a graceful quality degradation under packet losses.

Grace’s contribution is three-fold. First, we show that by simulating packet losses during autoencoder training (§3), it is possible to train autoencoders that learn not only how to compress frames efficiently but also how to code them in ways resilient to lost packets.3

Second, Grace presents a framework to stream autoencoder output over a sequence of frames (§4). Grace uses custom strategies to encode sequences of I-frames and P-frames and allow a receiver to determine when to decode a frame anytime after receiving the first packet, in order to balance frame delay and decoded video quality even when packet losses last for multiple frames. Grace can work on existing congestion control algorithms, which decide how fast packets should be sent, while Grace decides what packets should be sent.

Third, autoencoders are not as optimized (in speed and coding efficiency) as heavily engineered traditional codecs. To make it practical, Grace provides optimizations to speed up the encoding/decoding, reduce compute and memory overheads, and adapt encoding bitrates under dynamic bandwidth (§4.5). Importantly, Grace does not improve coding efficiency (higher quality with fewer bits), which most heavily engineered video codecs strive to optimize. Indeed, Grace’s video quality is still slightly lower than H.265 at the same bitrate.

We are not the first to apply autoencoders to data communication. Autoencoders have already been used in wireless and data communication to compress images, video files, and signals [28]. Notably, a recent work also uses custom autoencoders to obtain highly efficient scalable video coding (SVC) [27]. That said, Grace, to our best knowledge, is the first to develop custom autoencoders that realize data-scalable video delivery. The contribution of Grace is an end-to-end real-time video delivery framework, which tackles various challenges arising from training and using data-scalable autoencoder in real-time video communication.

We test Grace with traditional loss-resilient schemes and codecs on real and synthetic network traces, as well as various genres of videos. Our results show that while Grace’s compression efficiency is slightly worse than heavily engineered video codecs, it significantly reduces tail frame delay (by 2× at the 95th percentile when prior work has to retransmit packets or skip frame), with the marginally lowered video quality. To put it in perspective, Grace can significantly reduce the chance that frames arrive after user-perceivable delay (e.g., 200ms for video conferencing) and improve playback smoothness (higher frame rate).

This work does not raise any ethical issues.

2 Loss tolerance for real-time video

Many recent studies (e.g., [34, 43, 45, 58]) have shown that intermittent congestion, packet jitter, and packet drops widely exist and often cause long tail delay and low frame rate in real-time video applications. Consequently, a client may not be able to receive all packets of a frame before the frame is due for decoding (e.g., 40ms after the last frame was decoded if the video is 25fps). Thus, the ability to decode a frame, despite packet losses (including those dropped or those delayed by congestion), is pressingly needed.

2.1 Prior loss-resilience schemes

Loss resilience has long been an important research topic. While the details vary greatly among different loss-resilient techniques, it helps to view them through the lens of the high-level abstractions (summarized in Figure 2) that characterize how packets (coded data) are encoded and how decoded video quality of a frame changes with received data (packets). These abstractions also influence the interaction with the remaining parts of a real-time video application, such as congestion control and bitrate adaptation.

Retransmission-based clients encode a video frame by a traditional codec (e.g., H.264) with a target bitrate and must retransmit lost packets until a pre-defined set of packets (pre-determined during video coding) is reliably received. There are two typical retransmission-based schemes.

Basic TCP-based delivery must retransmit every lost packet for a frame to be decodable. Once all packets are received, the quality will be equal to the original encoded quality.

Scalable video coding (SVC) encodes a video at multiple quality layers such that a quality layer can be decoded as long as packets of this layer and all lower layers are received.
Forward Error Data Coding Correction Insertion Delay (msec) Kbps Scalable Video (FEC) 200 300 500 600 1200 1600 2000 Scalable based (Redundancy-based ones (FEC, intra-MB) tolerate a pre-defined maximum number of packet losses by adding redundancy (thus lowering quality when all packets are received). In contrast, data-scalable delivery decodifies a frame from any non-empty set of packets, improves quality with each new packet, and has high quality when all packets arrive. However, one single lost packet of a lower layer prevents all higher layers from being decoded.

With retransmission, video codecs can focus on coding efficiency (i.e., maximal quality under a given target bitrate). But it also suffers high tail delays when re-sending a packet takes longer than the jitter buffer and frame interval, causing a visually perceivable lag or a skipped frame. Even if the base (lowest) layer is padded with redundancy (e.g., FEC, explained shortly) [52], any packet loss can still be retransmitted to reach higher quality layers.

Redundancy-based schemes are used in complement to retransmission-based techniques. With this technique, a frame is decodable when at least $N\%$ packets are received. However, the decoded quality is lower with a smaller $N$. Here, $(1 - N\%)$ represents the redundancy rate. Redundancy-based schemes essentially trade coding efficiency for loss resilience. Redundancy-based schemes have been studied intensively.

Forward error correction (FEC) and Reed-Solomon and Fountain Codes [41] are often used in real-time applications. Ideally, these FEC schemes take an encoded video/frame and code it with redundancy bits, and the receiver can recover the data if any packets contain the same amount of bytes as the original data that was received.

FEC is known to have the “cliff effect”. No matter how much redundancy is added, the decoder has to receive at least as many bytes as in the encoded video file, i.e., retransmission cannot be avoided if the encoding video bitrate already exceeds the network capacity. Moreover, the decoded video quality is pre-decided by the video encoder and remains the same even if more data arrives. Thus, with too little redundancy, the receiver must wait for more packets to arrive or be retransmitted; with too much redundancy, the video quality will be low. Thus, they are most effective, if the sender knows how many bytes can be timely delivered in advance.

Unfortunately, it is not always possible to predict the number of missing packets before the receiver decodes a frame. Figure 3 gives an example showing why setting the FEC redundancy rate is difficult. We run WebRTC against a simple bandwidth trace over a network link of RTT 300 ms and a drop-tail queue of 1024 packets. When bandwidth suddenly drops, the sending rate and the FEC redundancy rate adjust only after any sign of congestion is returned to the sender (after at least one RTT), causing many packets to be dropped and delaying subsequent frames. The long delays could have been mitigated, if FEC adapts more conservatively (e.g., increases redundancy faster but decreases it more slowly). But this will cause lower video quality as more bytes are used for redundancy when bandwidth is stable. Techniques like dynamic reference selection [30] may mitigate this issue, but could not have avoided the congestion and delay of the first few frames. This corroborates the findings in the literature (e.g., [6])

Source-coding-based schemes (e.g., intra-MB insertion) do not explicitly add redundancy like FEC. Instead, it reduces the inter-dependencies between macroblocks (MBs) that are potentially similar to each other. For instance, intra-MB insertion adaptively sets more MBs in INTRA mode, making these MBs and others that refer to them decodable, even if other data is lost. Both intra-MB insertion and FEC must lower

---

4 Usually around 50 milliseconds in WebRTC
5 FEC is not in the standard of WebRTC. Our evaluation uses the FEC logic implemented in Google’s open-source repository [19].
coding efficiency: FEC adds redundancy data and intra-MB insertion keeps more inter-macroblock redundancy.

In short, retransmission-based schemes are sufficient if the RTT between the clients is low enough to conceal the retransmission delay. Redundancy-based schemes are suitable for connections with largely predictable packet loss rates. However, both of them fall short under high RTT and highly dynamic packet jitters and losses.

### 2.2 A new abstraction: Data-scalable delivery

This work envisions a new abstraction of loss resilience, data-scalable delivery, and presents a concrete implementation (§3). Data-scalable delivery should meet two criteria. (i) First, the frame is decodable once any non-empty subset of its packets are received, and quality improves with more packets received (or equivalently, the quality degrades gracefully with more packets missing). (ii) Second, unlike the redundancy-based approach, how a frame is encoded should not depend on the prediction of future packet losses. At a high level, this new abstraction can be expressed as follows:

\[
\text{Encode}(\text{frame}, \text{bitrate}_{\text{target}}) \rightarrow \{|\text{pkts}\}_{\text{sent}}
\]

\[
\text{Decode}(|\{|\text{pkts}\}_{\text{received}}|) \rightarrow \text{quality}
\]

**What’s new?** Figure 2 contrasts data-scalable delivery with other loss-resilient techniques.

- Unlike FEC (RS or Fountain Code), data-scalable delivery allows the receiver to decode after receiving any packet and obtain higher quality with each new packet received.
- While SVC might increase quality with more received packets, it is not data-scalable, because it imposes a hierarchical structure of packets. One missing packet may render multiple quality layers undecodable (even if the base layer is protected by FEC [50, 51]).

We should stress that any loss resilience comes at the expense of coding efficiency, and data-scalable delivery is no exception. We expect that the quality in absence of any packet losses will be slightly lower than traditional codecs.

### 2.3 Why autoencoder?

Our approach to data-scalable delivery is based on autoencoders, an emergent class of neural-network coders for video (I-frames and P-frames). The recent autoencoders borrow ideas from traditional codecs (e.g., compressing motion vectors and residuals separately, rather than compressing each frame from scratch), and replace handcrafted heuristics with multi-layer neural nets (NNs) which are trained on large sets of videos (e.g., Vimeo-90K [55]). Recent autoencoders’ performance has significantly improved [40, 49].

We choose to use autoencoders not because of their coding efficiency (which comes with caveats discussed shortly), but because their quality under data erasure (albeit suboptimal) might fit the definition of data-scalable delivery. Figure 4 shows an example: we use a pre-trained P-frame autoencoder [40] to encode frames from a video (not from the training set) and test its decoded quality with an increasing rate of random data erasure (we will explain how this simulates packet losses in §3.2). The red curve shows that as more data is received, the decoded quality gradually improves.

However, as the figure shows, the pre-trained autoencoder degrades quite sharply with more data erasure. (After all, these autoencoders are not trained to perform well under random data erasure.) In §3, we will show how to train data-scalable autoencoders that produce better quality under various packet losses (e.g., the blue line in Figure 4).

### 2.4 Caveats of using autoencoders

Despite being emergent and not as heavily optimized as traditional codecs, autoencoders have gained increasing attention in the signal processing community for image compression [21, 31] and wireless communication [26, 28], by the multimedia community for high-volumetric video compression [37, 38], and recently by the networking community for chunk-based video streaming [27], though these efforts still rely on traditional schemes (like FEC) to handle packet losses.

Though autoencoders are studied extensively, there has been much less attention given to a thorough comparison between autoencoders and traditional codecs (H.265 and H.264, which have been extremely optimized in speed and efficiency). This could lead to potentially biased (and sometimes exaggerated) perceptions of autoencoders’ performance. For instance, if one tests an autoencoder and H.265 under the same small (4-10) GoP, autoencoders will outperform H.265 because small GoPs align better with how autoencoders are trained but not with H.265. These comparisons could also be unfair to autoencoders as well. For instance, H.26x searches optimal reference MBs from multiple frames to reduce residuals, whereas most autoencoders derive motion vectors and residuals from a single reference frame (which is not a fundamental constraint) and instead, they focus on compression of given motion vectors and residuals. Moreover, autoencoders can be trained to optimize either PSNR or SSIM, so papers evaluate only the metric that the model is trained for.

This work reports our efforts to make autoencoders loss re-

![Figure 4: Loss resilience of an example pre-trained autoencoder (red). It is already data-scalable (decodable with any subset of packets), but the quality drops much faster than the graceful degradation observed in our solution.](image-url)
silient (§3.4) as well as practical (in speed and memory usage) for real-time videos (§4.5). In our evaluation of autoencoders, we have tried our best to avoid pitfalls (e.g., those listed in [7]). (For space limitation, we elaborate the details in §5.1.) For instance, we report both PSNR and SSIM and use H.26x’s default encoding setting without limiting GoP; and stress-test autoencoders on video sets different from the training set. We also use a common setting of real-time video [1, 25] with the fast preset in H.265 and no B-frames or rc-lookahead.

We hope to present a balanced view in this work: even though the autoencoders have lower coding efficiency than the heavily engineered video codecs like H.265, we believe that their potential to reduce tail delay (via data-scalable delivery) outweighs the slightly lowered quality, which is also being constantly improved by better neural-network models.

3 Grace-AE: Data-Scalable Autoencoder

At a high level, Grace uses a new data-scalable autoencoder, Grace-AE and a custom delivery framework that leverages the new autoencoder (§4) to balance tail delay and quality.

3.1 Basic formulation of Grace-AE training

We denote the encoder (a neural network parameterized by weights φ) by Cφ and the decoder (a neural network parameterized by weights θ) by Dθ. For an input frame x (a c × w × h tensor), the coded data z = Cφ(x) will be sent to the receiver, which then runs the decoder to reconstruct the original frame from z: ŷ = Dθ(z). Note that the encoder and decoder performing the lossy reconstruction are separate from the lossless entropy coding or FEC coding. (We will discuss entropy encoding and packetization in §3.2.) We retrain the DVC autoencoder [40] to encode P-frames, and discuss how to add I-frames in §4.1. (We do not consider B-frames as they are not commonly used in low-latency videos.)

Conventional autoencoder training: An autoencoder is trained to minimize the “loss” function (not “packet loss”) F(ŷ, x) = Distortion(ŷ, x) + α · Size(ŷ), which is a weighted sum of its pixel distortion Distortion(ŷ, x) (L2-norm of ŷ − x)\(^8\) and its entropy-encoded size Size(ŷ) (e.g., estimated by a pre-trained neural net). A smaller α means that the output of Cφ(x) will tend to have better visual quality in Dθ(z) but also higher bitrate. (We present Grace’s bitrate adaptation in §4.5.) Since Cφ, Dθ, and F are differentiable, the encoder and decoder NNs can be trained as an end-to-end architecture on a training video set X:

\[
\max \mathbb{E}_{x \sim X} F(D_{θ}(z), x), \text{ where } z = C_{φ}(x) \quad (1)
\]

Training data-scalable autoencoder: The training process in Eq. 1 can be seen as having no packet loss, i.e., the decoder’s input equals the encoder’s output. To improve coding

\[\text{Simulating packet losses}\]

\[P_{\text{loss}} \text{ takes the encoder output } C_{φ}(x) \text{ and returns the distribution of the data seen by the decoder after packet erasures under some DMR. Simulating packet losses in Eq. 2 raises three unique questions, and we will address them next.}

1. How should \(P_{\text{loss}}\) convert \(C_{φ}(x)\) to data packets to simulate the effect of dropping a packet?
2. Given that the function \(P_{\text{loss}}\) may not be differentiable, how to train autoencoders with Eq. 2?
3. What packet loss rates should be simulated in training so that the trained model can handle unseen loss rates?

3.2 Simulating packet losses during training

Packetization and data erasure: First, to packetize the encoder output \(z = C_{φ}(x)\), Grace uses a packetization function \(A\) (illustrated in Figure 5) to map each element position in \(z = C_{φ}(x)\) (a 3-D or 4-D tensor) to one of \(k\) non-overlapping lists\(^9\) and then losslessly encodes the elements in each list to a bitstream of a packet. Conversely, depacketization first losslessly decodes each packet to a list of elements and maps elements back to their original positions in \(z\) by the inverse function \(A^{-1}\). One can use an off-the-shelf entropy coder to losslessly code the elements in a packet to a bitstream. With per-packet entropy coding, each packet is individually decodable. Per-packet entropy encoding is commonly used in intra-MB insertion (explained in §2.1) and other source-coding loss resilient schemes. We use arithmetic coding [9] to encode each packet. Because the elements in each packet share the same distribution as all elements in \(z\) (as we will show shortly), different packets share the same arithmetic coder (without needing a separate arithmetic distribution per packet), so each list has the same number of elements, the encoded packet size of the list will be roughly the same.

\[\text{Simulating packet losses}\]


\[P_{\text{loss}} \text{ takes the encoder output } C_{φ}(x) \text{ and returns the distribution of the data seen by the decoder after packet erasures under some DMR. Simulating packet losses in Eq. 2 raises three unique questions, and we will address them next.}

1. How should \(P_{\text{loss}}\) convert \(C_{φ}(x)\) to data packets to simulate the effect of dropping a packet?
2. Given that the function \(P_{\text{loss}}\) may not be differentiable, how to train autoencoders with Eq. 2?
3. What packet loss rates should be simulated in training so that the trained model can handle unseen loss rates?

3.2 Simulating packet losses during training

Packetization and data erasure: First, to packetize the encoder output \(z = C_{φ}(x)\), Grace uses a packetization function \(A\) (illustrated in Figure 5) to map each element position in \(z = C_{φ}(x)\) (a 3-D or 4-D tensor) to one of \(k\) non-overlapping lists\(^9\) and then losslessly encodes the elements in each list to a bitstream of a packet. Conversely, depacketization first losslessly decodes each packet to a list of elements and maps elements back to their original positions in \(z\) by the inverse function \(A^{-1}\). One can use an off-the-shelf entropy coder to losslessly code the elements in a packet to a bitstream. With per-packet entropy coding, each packet is individually decodable. Per-packet entropy encoding is commonly used in intra-MB insertion (explained in §2.1) and other source-coding loss resilient schemes. We use arithmetic coding [9] to encode each packet. Because the elements in each packet share the same distribution as all elements in \(z\) (as we will show shortly), different packets share the same arithmetic coder (without needing a separate arithmetic distribution per packet), so each list has the same number of elements, the encoded packet size of the list will be roughly the same.

\[\text{Simulating packet losses}\]
If a packet is lost, Grace will set each element whose position is mapped to the lost packet to zero. Thus the packetization function $A$ decides how missing one packet would affect the coded data seen by the receiver. Grace’s packetization function is a uniformly random mapping ($A$ pseudo-random function which has a reversible mapping$^{10}$). As a result, with a 30% packet loss rate, $P_{\text{loss}}(z|C_0(x))$ will produce a distribution of $z$ that randomly zeros 30% of elements in $z$.

**Why random packetization:** The choice of randomized packetization empirically works better than alternatives, for two reasons.

- In an autoencoder output $z$, some elements have more impact on the decoded quality (defined in §3.3). A random mapping thus makes sure that losing $x\%$ packets will affect $x\%$ important elements (whereas non-randomized packetization may affect more than $x\%$ important elements).

- A random mapping ensures that all elements in $z$ have the same distribution as those mapped to each packet. This makes it easier to maintain roughly the same entropy-encoded size of each packet.

Figure 6a(a) empirically compares Grace-AE trained with the random packetization and Grace-AE trained with alternative schemes, such as block-based packetization (i.e., each packet contains a contiguous block of elements in $z$) and interleaving-based packetization. Both are suboptimal, since important elements can be close to each other and have a fixed distance from each other under $z$‘s specific tensor structure, so in the worst case both block-based packetization and interleaving packetization can drop a higher fraction of important elements than packet loss rate.

**Choice of packet loss rate in training:** At first glance, we should simulate as many packet loss rates during training as possible, so that the model can deal with various packet losses. However, our empirical results (Figure 6(b)) suggest that adding too much randomness in $P_{\text{loss}}$ might make the training converge to suboptimal autoencoder models. Specifically, training Grace-AE with just a few values of packet loss rates generalizes better than with a wide range of packet loss rates, even on unseen test packet loss rates. Moreover, the packet loss rates in training should include 0% (otherwise the trained autoencoder will perform significantly worse than the pre-trained autoencoder on low packet loss rates.

**Making Grace-AE trainable:** Since $P_{\text{loss}}$ is a non-differentiable random function, the gradient of the expectation of $F$ in Eq. 2 cannot be directly calculated. To address this issue, we use the REINFORCE trick [35] for reparameterization. We express the gradient of Eq. 2 as

$$\nabla_{\theta} P_{\text{loss}}(z) = P_{\text{loss}}(z) \nabla_{\theta} \log P_{\text{loss}}(z)$$

where $P_{\text{loss}}(z)$ is our packet loss distribution so our gradient of the expectation of $F$ becomes

$$\nabla_{\theta} \mathbb{E}_{x \sim P_{\text{loss}}(z)} ([F(D_0(z), x)]) = \mathbb{E}_{x \sim P_{\text{loss}}(z)} ([F(D_0(z), x) \nabla_{\theta} \log P_{\text{loss}}(z)]) \quad (3)$$

which can be estimated using Monte-Carlo sampling $\approx \frac{1}{N} \sum_{i=1}^{N} F(D_0(z_i), x) \nabla_{\theta} \log P_{\text{loss}}(z_i)$ Since in our application the loss is independent and identically distributed random variable, the gradient evaluates to either 0 or 1, hence we propagate the gradients for the encoder only for $F(D_0(z), x)$ where $P_{\text{loss}}(z_i) = 1$.

### 3.3 Why is Grace-AE data-scalable?

We believe that the reasons are two-fold. First, compared to SVC, FEC, or H.26x-based coding, the elements in Grace-AE’s encoding output (before entropy encoding) do not impose any strict structure (e.g., hierarchy or dependencies). Instead, the relationship between elements in Grace-AE output is “flat”, and changing any element (to zero) would not render the frame non-decodable.

Grace-AE’s decoder can be seen as a special case of variational autoencoders (VAE) [48], which are generative neural networks trained to create realistic images from a random point in the input space. In the case of Grace-AE, our random point follows the distribution of $P_{\text{loss}}$.

Second, compared to the pre-trained autoencoder, individual elements in Grace-AE’s output have a lower impact on the decoding quality, so quality degrades more gracefully with more lost data. Though the NNs are too complex to show this directly, we can indirectly illustrate it as follows. For a frame, we first calculate the “gradient” of decoding quality with respect to each element in $z$. gradient $= \frac{\partial F(D_0(z), x)}{\partial z}$. Then, we sort the elements in descending order of their gradients. As more top $k$ elements are zeroed, we observe that the quality of the pre-trained autoencoder drops sharply, whereas that of Grace-AE drops much more slowly (not shown for space limitation).

### 4 Real-time video delivery over Grace-AE

Grace is built on the data-scalable autoencoder Grace-AE. We focus on unique issues arising from the use of Grace-AE...
as the codec, including how to deliver each frame to strike
a desirable delay-quality tradeoff, and how to deliver a se-
queness of frames when congestion or packet losses affect
multiple frames. The choice of the congestion control logic is
complementary to Grace-AE. In §5.3, we will evaluate Grace
with different congestion control logic, such as GCC in WebRTC [22] or Salsify CC [30]. We do not claim that Grace’s
strategies are new (e.g., reference frame synchronization has
been used in [30]). Nonetheless, Grace integrates concrete
design choices driven by the need, in order to unleash the full
potential of Grace-AE and its data-scalable codec primitive.

4.1 Encoding a sequence of frames

Placement of I-frames poses a particular challenge for Grace’s
autoencoders (both pre-trained and re-trained), for two rea-
sons: (1) I-frames are generally bigger in size than P-frames,
and (2) while the first few P-frames after an I-frame enjoy
high quality, quality gradually degrades after about 10 frames.

A strawman would add I-frames frequently (say every 10
frames)\(^1\). Even though this might achieve a decent tradeoff
between quality and average bitrate (comparable to H.264 and
H.265 with large I-frame intervals), sending I-frames (which
have bigger sizes than the P-frames in between) frequently
will make it difficult for congestion control to send the frames
at a fixed interval (inversely proportional to the frame rate).

To add I-frames frequently without the intermittent bitrate
spikes caused by the I-frames, Grace encodes a small patch
(about 128×128 to 512×512) as a tiny I-frame, which we
call I-patch, on every frame (an I-patch will not impact the
original P-frame), and the location of the I-patch on a frame
changes over time such that the I-patch will scan through the
whole frame size every \(k\) frames. Depending on the original
frame size, \(k\) varies between 6 and 20. This scheme effectively
“amortizes” the size of I-frames across \(k\) frames, thus smoothing
the frame size as well as quality across frames (illustrated in
Figure 7). Moreover, since the I-frame autoencoder’s en-
coding delay is proportional to the input size, encoding a
small I-patch adds only a small compute overhead for each frame.\(^2\)

4.2 Packet losses across multiple frames

An important benefit of adding an I-patch (a tiny patch en-
coded as an I-frame without referring to previous frames) at
a different place in each frame is: each packet will be encoded as
an I-patch every \(k\) frames. This has a significant implication
for loss resilience over multiple consecutive frames.

\(^1\) Many computer-vision papers on autoencoders have conveniently used
small GoP when compared against H.265, but the implication of large
I-frame size for congestion control is rarely discussed.

\(^2\) We should clarify that design choice of frequent I-frames works well for
autoencoders, but not for classic codecs for two reasons. First, an I-frame
generated by our autoencoder is only 2-5× bigger than a P-frame, whereas
this ratio in H.264 is 4-10×. Second, the P-frame quality in H.264 is relatively
stable, whereas the quality of a P-frame is high following an I-frame (or an
I-patch).

An acute reader may realize a potential problem arising
from encoding and decoding being asynchronous. For in-
stance, when the sender encodes the 10\(^{th}\) frame as a P-frame,
it needs the decoded the 9\(^{th}\) frame as the reference (to compute
motion vectors and residuals), and without knowing which
of the 9\(^{th}\) frame’s packets will be lost, it will assume that the
9\(^{th}\) frame is decoded with all packets being received. (This
is common to classic codecs and autoencoders.) However, if
the receiver decodes the 9\(^{th}\) frame with only a subset of its
packets, the actual reference frame used to decode the 10\(^{th}\)
will differ from what the encoder has assumed. Thus, most
codes require all packets of the 9\(^{th}\) frame to be received before
decoding it (thus longer delay), or let the sender encode
the 10\(^{th}\) based on an earlier reference (say the 5\(^{th}\)), whose lost
packets are already known to the sender. The latter synchro-
nizes the encoding and decoding states (like in Salsify [30]
during packet losses), but it also inflates the frame size, since
the difference between the 10\(^{th}\) and the 5\(^{th}\) frames is much
greater than between the 10\(^{th}\) and the 9\(^{th}\) frames.

Grace uses a simple yet efficient solution—it does not al-
ways synchronize the encoding/decoding state on every single
frame; instead, it relies on the frequent I-frames (explained in §4.1)
to synchronize the encoding/decoding states in each
pack-size region. This works for two reasons. First, without
synchronizing the encoding and decoding states, Grace-AE’s
quality degradation increases only marginally when multiple
consecutive frames (less than about 10) are affected by packet
losses. For instance, applying a 50% packet loss rate on
one single frame can reduce PSNR by 3-4, and applying the
same loss rate on 5 consecutive frames reduces PSNR by
7-9. Second, each packet will be encoded as an I-patch every \(k\)
frames (\(k\) is between 6 and 20), so the impact of packet losses
will not last for more than \(k\) frames if the next I-patch is not
lost. That said, in the worst case, if the I-patches of the same
region are dropped persistently, this could cause bad decoded
pixel values (i.e., a small patch of white pixels), though we
have observed it extremely rarely.

4.3 Choosing when to decode a frame

The data-scalable abstraction allows the receiver to flexibly
decide, as packets arrive at the receiver, whether to decode the
frame using currently arrived packets or wait for more packets
in hope of improving the quality. (In contrast, with FEC, the
receiver will always wait until enough data is received.)

To make this decision, Grace defines the utility of a

![Figure 7: Encoding each P-frame with a small I-patch leads to smoother frame sizes than naively inserting I-frames.](image-url)
Arrival of the

frame as a linear combination of decoding quality and frame delay (since the natural decoding time) on each frame: 

\[ Q(|\{\text{pkts before } t\}|) - \beta \cdot t, \]  

where \( t \) is when the frame is decoded. \( Q(n) \) is the frame’s decoding quality with \( n \) packets. The precise values of \( Q(n) \) are frame-dependent, but they are roughly similar across frames, so Grace estimates \( Q(n) \) using history frames. This objective works well in practice, though there might be other objectives that better capture users’ quality perception. An optimal design of such utility is beyond the scope of this paper.

With this utility in mind, Figure 8 illustrates Grace’s logic: on receiving the \( i \)th packets of a frame (not necessarily the \( i \)th sent packet) at time \( t_i \), the receiver will immediately decode the frame if all packets of the frame have arrived; otherwise, it will wait for the next packet of the frame until a deadline \( t^* = t_i + \frac{1}{\beta} \cdot (Q(i + 1) - Q(i)) \), and if it has not received the \((i + 1)\)th packet by then, it will decode the frame with \( i \) packets. The reason is that if the next packet comes at time \( t_{i+1} \) after \( t^* \), the utility \( Q(i + 1) - \beta \cdot (t_{i+1} - \tau) \) will be lower than the utility \( Q(i) - \beta \cdot (t_i - \tau) \) if the receiver does not wait at all. This strategy makes intuitive sense: the receiver will wait longer if the improvement in quality by getting one more packet (\( i.e., Q(n + 1) - Q(n) \)) is higher, but it will not wait for too long.

### 4.4 WebRTC integration

Grace is implemented with 3K lines of code, in both Python (mostly for autoencoder NNs) and C++ (for frame delivery and WebRTC integration). The code and trained model of Grace will be made public upon the publication of this paper. The integration with WebRTC is logically straightforward since Grace-AE (including I-frame and P-frame encodings) exposes similar interface as the default codec in WebRTC.

We substitute the libvpx VP8 Encoder/Decoder in WebRTC with our Grace-AE implementation. When the sender encodes a frame, it parses the image data from the VideoFrame data structure (YUV format) into torch.Tensor (RGB format) and feed it into our Grace-AE encoder, which will return the encoded result as a byte array. Then the encoded bytes are stored into an EncodedImage (class in WebRTC) and sent through the network to the receiver as RTP packets. We modify the built-in RtpVideoStreamReceiver (class in WebRTC) so that the receiver could flexibly decode the received packets even when not all the packets are received. When the receiver decides to decode the frame, it depacketizes the received packets into encoded data. Then it will use the Grace-AE decoder to decode the image into RGB format and then convert it back to YUV for displaying on the receiver side.

#### 4.5 Optimizations

While Grace-AE is functionally similar to traditional codecs, it faces a few technical challenges to function as the codec in a real-time video client. Note that these challenges are not specific to loss resilience, so we do not consider packet losses in the rest of the section.

**Reducing compute overhead:** Most research on autoencoder focuses on coding efficiency (using fewer bits to get higher quality), but the encoding/decoding speed is much lower than the heavily optimized codecs like H.265. Fortunately, we show that on a single NVIDIA GeForce 3080 GPU, Grace’s encoding speed can be greatly improved: from 5 fps on 720p HD (and 11 fps on 480p) videos by the off-the-shelf implementation to 18 fps (and 40 fps). More results in §5.4. Grace achieves such speedup by downsizing the image (by default 4x) during motion-vector estimation and motion compensation, and then upsamples the motion-compensated image before calculating and encoding the residuals. This speeds up encoding because motion-vector estimation and motion compensation are the slowest steps during P-frame encoding and their delays are proportional to the input size. Thus, the overall encoding delay will be reduced if they are fed with downsized images. At the same time, because the residuals are still computed between the full-resolution image and the motion-compensated image, the coding efficiency is not affected by this change for most frames.

**Bitrate control:** In real-time video, throughput estimation can vary even between two consecutive frames, and the encoded frame size should be barely below the target frame size. This has been a challenge with traditional codecs. Grace provides bitrate control logic similar to [30] to encode a frame with a maximum size below a target. We train multiple autoencoders, each using a different weight of frame size \( \alpha \) in its training loss function (§3.2). These resulting autoencoders cover a range of bitrate range. As long as the target bitrate is within this range, Grace should be able to select the autoencoder whose encoded frame size is barely lower than the target. A naive solution will run multiple autoencoders (like [30]), but the compute overhead will increase linearly. Fortunately, we can again re-use the motion-compensated image to get the residuals, and only run different versions of residual encoders to encode the residuals with different bitrates. Since residual encoder is faster than the motion estimation and compensation, the speed up is significant.

**Reducing memory footprint:** For the client to switch be-

---

11This is because \( Q(i + 1) - \beta \cdot (t_{i+1} - \tau) < Q(i + 1) - \beta \cdot (t^* - \tau) = Q(i) - \beta \cdot (t_i - \tau) \)
tween multiple autoencoders (with different bitrates), their memory footprint should be minimized, so more of them can be loaded into GPU memory to avoid the delay of loading and unloading models. Grace trains multiple autoencoders by fine-tuning the last 10-25% layers of the same model (i.e., they share the backbone layers), so only the one autoencoder and the last few layers of different autoencoders need to be loaded in GPU memory. This reduces the memory footprint of the models by 60% compared to when all models are trained separately without sharing any layers. A similar idea has also been used in Swift [27] in multiple SVC layer encoders.

5 Evaluation

The findings of our evaluation can be summarized as follows:

- **Loss resilience:** Though Grace-AE’s quality under no packet loss is on par with H.264 and slightly worse than H.265, it is data-scalable—graceful quality degradation with higher loss rate—under various videos and bitrates.
- **Quality-delay tradeoff:** On real and synthetic network traces with bandwidth fluctuation, Grace reduces tail delay by 2×, compared to other loss-resilient schemes (FEC, Salsify) while still obtaining decent quality.
- **Overhead:** Our implementation of Grace can encode/decode 480p video at 40 fps/50 fps and 720p HD video at 18 fps/30 fps, a 1.5× speedup compared to off-the-shelf autoencoders.

5.1 Setup

**Baseline implementations:** Our baseline codecs include H.265 and H.264. We use their implementations in ffmpeg (version 4.2.7). Importantly, we do not restrict their GoP (I-frame interval) and the number of history frames to search for reference macroblocks. Following recent work in real-time video coding [1, 25], we use the zerolatency option (no B-frames) and the fast preset of H.265

We use implementations for SVC, Salsify, and FEC, based on H.265. We simplify the implementation in a way that makes their quality slightly better than they would actually do. Our implementation of SVC assumes that each quality level exactly matches the quality and size of the non-SVC coding of H.265, whereas even the state-of-the-art SVC (based also on autoencoders [27]) achieves lower quality than H.265 at a quality level of the same frame size. (This effectively is an idealized version of Swift [27].) We follow Salsify’s logic for congestion control, frame skipping, and packet pacing, but instead of using Salsify’s multi-trial encoding to approximate the target bitrate of each frame, we assume that the encoded frame size exactly matches each frame’s target bitrate but the quality matches the qp value whose bitrate is barely higher than the target bitrate. For FEC-protected H.26x, we use the same loss-rate prediction logic implemented in the public release of WebRTC [19] but remove the upper-bound of 50% redundancy rate that is used in WebRTC e.g., if the predicted packet loss rate is 60%, we assume FEC will tolerate any packet loss ratio under 60%. While these simplifying assumptions lead to a slightly optimistic estimate of the baselines’ performance, we hope they make our results more relevant in the long run, as the implementations of SVC, and bitrate control, and FEC might evolve and improve in the future.

**Network and compute conditions:** We test Grace on 8 real bandwidth traces distributed with the Mahimahi network-emulation tool [47]15. By default, we set the one-way network propagation delay to be 100ms and the queue size to be 25 packets. We also show the impact of different propagation delays and queue sizes in §5.3. We conduct all experiments on a machine with two Nvidia GeForce RTX 2080 GPUs as the sender and receiver (each using one GPU). The server runs Ubuntu 18.04 with 2 Intel Xeon Silver 4110 CPU and 64GB memory. We measure Grace’s encoding and decoding delays on the GPU (see results in §5.4) and we include these delays in Grace’s frame delay.

**Metrics:** We measure the performance of a video session in three aspects. Quality of a frame is measured by SSIM and PSNR (RGB and YUV), and we report the average value across received frames. Delay of a frame is measured by the time lapse between the beginning of encoding to the end of decoding. For skipped frames (by Salsify), their decoding time equals the next decoding time of the next received frame. We report the tail (95th percentile) of the delays across frames in a session.

**Test videos:** Our evaluation uses 60 videos from three public datasets (Kinetics [23], UVG [44] and FVC [18]), each video is of at least 10 seconds. Importantly, we choose these videos because they are different from the training set, allowing us to stress-test the generalization of autoencoders. The FVC dataset includes 5 videos captured from video conferencing, one of Grace’s main target use-cases. These videos cover a range of spatial and temporal dynamics (§A.3) and cover three resolutions (640×360, 1280×720 and 1920×1280), with at least 5 videos in each resolution. The performance is reported as an average across all the frames in the videos.

5.2 Coding efficiency and loss resilience

Figure 9 compares Grace-AE with the baselines codecs16 under different packet loss rates when encoding videos of the same resolution and same target bitrate. In each figure, all lines have almost the same encoding bitrate with difference...

14The command line we used to encode a video is ffmpeg -y -i Video.y4m -crf 0 -keyint 3000 -x265-params "crf=Q:keyint=3000" output.mp4 where Q controls the quality and size of the output video

15The data set has 16 traces. We filtered out the traces having an average bandwidth lower than 1Mbps as they do not fit the bandwidth requirement of video traffic.

16We do not put Intra-MB insertion baseline here, since it’s quality will quick drop below x-axis when loss ratio is between 5% to 10%. 

9
controlled within 5%. We ensure that Grace-AE’s encoded bitrate is never higher than the baselines, so even if the actual encoded bitrate is different, it does not favor Grace-AE.

The most salient feature of Grace-AE is the graceful quality degradation with more packet losses. The average PSNR drop is only 2.1 to 3.9 under 30% and 50% packet losses. Even when the loss ratio is as high as 80%, it has an average PSNR of 36.4, which is much better than the baselines except H.264 with 80% FEC. Compared with H.264 with fixed FEC protection, Grace-AE can have a higher PSNR of 1.1 to 2.0 among different resolutions and different bitrates when there is no loss. This is because any arrived bytes in Grace-AE contribute to the quality while in FEC, the redundant data is useless even if they do arrive at the decoder. On the other hand, FEC-based loss resiliency is also in a dilemma that a low FEC rate cannot protect against the high loss, but a high FEC rate leads to poor quality when the packet loss ratio is small. Our Grace-AE solves such dilemma by the graceful quality degradation. Grace-AE is also better than SVC with unequal FEC protection (§2.1). The quality of SVC decreases quickly when the loss rate is small, as the high-quality layers are unlikely to be decodable in such cases. We also note that the pretrained autoencoder models, which have not seen any losses in training, degrade poorly with higher loss rates.

For any loss-resilient schemes, the ability to tolerate packet losses often comes with a drop of quality in absence of packet losses, and Grace-AE is no exception. Figure 10 compares the

Grace-AE’s quality under no packet losses with that of H.264 and H.265 across the test videos (all of which are randomly sampled from three datasets that are different from the training set, §5.1). Fortunately, at the same encoded bitrate, Grace-AE is similar to H.264 and is only slightly worse than H.265. When bitrate varies from 2Mbps to 8Mbps, the average PSNR is 44.39, 44.31, and 44.84 for Grace-AE, H.264 and H.265 respectively.

Remember that H.26x uses the zerolatency option and fast preset (which is typical to real-time videos, including WebRTC). We confirm that without these settings, H.265’s coding efficiency is better than Grace-AE in the absence of packet losses. However, our goal is a data-scalable autoencoder for real-time videos (shown shortly), rather than an autoencoder that outperforms heavily engineered H.26x codecs. Moreover, the coding efficiency of autoencoder could still be improved (e.g., our autoencoder estimates motion vectors based only on one reference frame).

5.3 Video quality vs. tail frame delay

Next, we run Grace and the baseline codecs on real bandwidth traces from Verizon and T-Mobile LTE (§5.1). We set RTT at default values (200ms) and drop tail queue length of 25 packets. We implemented two different congestion control

\[ \text{usageType} = \text{CAMERAVIDEO_REALTIME} \] (similar to zerolatency) and \[ \text{complexityMode} = \text{LOW_COMPLEXITY} \] (similar to fast)
algorithms: Google congestion control [22] (GCC) and the congestion control from the previous work (Sal-CC) [30]. We also implemented FEC-protected H.265 and Salsify codec as the baseline codecs (§5.1).

Figure 11 compares Grace and the baselines, in average quality (both PSNR and SSIM) and the 95-th percentile of frame-level delay across frames in a session. We see that Grace significantly decreases the tail frame delay: Grace is 251 ms and 267 ms when using GCC and Sal-CC respectively, compared to 401ms and 412ms for Salsify codec, and 406ms and 1652ms for H.265 codec. When there is a packet loss, Salsify codec needs to skip frames, causing a delay increase. For H.265 codec, it needs to wait for those lost packets being retransmitted, and such retransmission traffic will compete with the normal video traffic, causing a higher delay on H.265.

When under the same congestion control (Sal-CC or GCC), the average PSNR of Grace, Salsify codec and H.265 are 46.2, 47.8, 47.9, respectively. Though Grace’s PSNR is on par or slightly lower, we believe the reduction in tail frame delay is worth the minor drop in quality. We make a similar observation when measuring quality in SSIM (dB): Grace’s quality is similar or slightly lower than the quality of baseline using the same CC.

We then test Grace’s performance on the same traces but under different queue lengths and different propagation delays. Figure 12 and Figure 13 show that Grace consistently reduces tail delay under different network delays or under different droptail queue lengths. In particular, Grace has more delay reduction when the queue is shorter. This is because during congestion, shorter queues tend to drop previous frames packets more aggressively, and might still delivery some packets (though not all) of the latest frame to the receiver. In such cases, Grace lets the client decode the frame early based on any subset of packets it has received.

Finally, we show how Grace and the baselines behavior in a concrete bandwidth trace sample (Figure 14). The bandwidth drops from 5Mbps to 1Mbps at 1.9sec and last for 200ms, before bouncing back to 5Mbps (another bandwidth drop occurs at 3.9sec). We can see that during each bandwidth drop, Grace’s delay does not increase as sharp as the baselines (notably Salsify is the second best since it could skip frames). Grace and Salsify use the same CC, so their quality roughly matches on frames not skipped by Salsify. That said, Grace’s quality degrades marginally with no frame skipping during congestion, and it bounces back once packet losses disappear.

5.4 Memory and computing overhead

Our implementation of Grace-AE optimizes both memory footprint and encoding/decoding delay (described in §4.5), and we test its effectiveness on a Nvidia GeForce RTX 3080 GPU. Figure 15a and Figure 15b show that compared to the original (unoptimized) implementation of pre-trained autoencoder (which takes 100ms to encode a 480p frame), our implementation can encode a 480p frame in less than 25ms (a 4× speedup). This translates to almost 40fps. Even for 720p HD videos, our optimized codec can achieve 18fps (55ms encoding per frame). This acceleration comes from downsampling frames in motion-vector estimation. Moreover, when adapting to a target bitrate, Grace encodes the next frame multiple times without needing to run the entire encoder multiple times, because it re-uses the motion-compensated prediction, so only the residuals need to be encoded at multiple bitrates. Figure 15c shows that the delay of encoding a frame at three different bitrates only increases marginally compared to encoding at the bitrate for the original frame. Grace-AE is one of the few auto-encoder implementations that can be used in real-time applications, and it achieves comparable encoding/decoding speed to traditional codecs like H.265 and Salsify.
coding one frame.

In terms of memory usage, our optimized implementation saves the memory footprint of loading 9 models by over 60% since most NN layers are shared. Note that Grace-AE does not change the NN architecture of the autoencoders—only the weights are retrained and the way they are used is optimized. We speculate that they could be made more efficient with appropriate modification to its NN architecture.

6 Related work

Loss resilience: Loss resilience techniques fall into three categories. Channel coding includes FEC techniques, Reed-Solomon code, LDPC and fountain codes [41, 42]. While the (ideal) ability to reconstruct video of bitrate $k$ from any received data of bitrate $k$ is appealing, it means that some packets must be delayed/retransmitted if the encoded video bitrate already exceeds the link capacity, so using channel coding alone is not ideal for real-time video applications [29]. Source coding adds redundancy during the video coding process, such as Intra-MB insertion [24]. A more promising approach to loss-resilient video coding is joint-source-channel coding, and our Grace-AE belongs to this category. The closest related work is DeepJSCC [36] which trains an autoencoder to code images in a representation that is both compact and robust to signal noises [21, 31]. Grace-AE differs from DeepJSCC on two key fronts. First, they target physical-layer protection against signal noises. While physical-layer noises can be modeled as differentiable linear transformations (as in [21, 31]), packet drops (DMR) are not differentiable and need to be handled differently (see §3.2). Second, unlike compressing individual images, frames in a streaming video cannot be treated separately, otherwise any error on one frame may propagate to future frames.

Video compression: Obtaining loss resilience must sacrifice coding efficiency (lower quality in absence of packet losses), and Grace-AE is no exception. Most commercial video coding techniques are based on classic codecs, such as H.265 and VP9. Though video autoencoders (e.g., [40, 49]) have greatly improved their coding efficiency, it remains unclear if they outperform heavily engineered state-of-the-art video codecs, which do not limit the search for optimal reference MB and can run multiple passes on a video to optimize coding efficiency. Our evaluation focus on a specific setting (H.26x without B-frames and the “fast” preset) and shows that Grace-AE’s coding efficiency is comparable with H.26x. In a recent work [27], autoencoders’ coding efficiency is also shown to be comparable with H.265. That said, we choose to build Grace on autoencoders not for its coding efficiency, but for its potential to realize data-scalable delivery.

Adaptive video streaming: There has been intense research on adaptive streaming, including adapting sending rate (e.g., [53, 58]), encoding bitrate (e.g., [30, 54, 57]) and FEC redundancy rate (e.g., [29, 32, 46]). These congestion control and bitrate adaptation algorithms are essentially reactive to intermittent congestion and packet losses. Grace is complementary to them in that data-scalable delivery eschews the need for accurate congestion and packet loss prediction and instead gracefully degrades their quality with more packet losses.

7 Limitations

At least one packet has to arrive: Grace allows the receiver to choose when to decode after the first packet arrives, but it cannot decode a frame with no received packet. This means Grace will not help during network disconnection or severe congestion that blocks any packet arrivals. One potential way to avoid such extreme situation is to leverage multiple concurrent connections and run Grace over multipath TCP (with custom packet scheduling to discard or deprioritize untimely packets.)

Bounded by autoencoder’s performance: Though §5 shows that Grace-AE’s compression efficiency is comparable to H.264 under no packet loss, its quality-bitrate tradeoff is bounded by that of the pre-trained autoencoders. This is expected, since we choose not to modify the autoencoder architecture, and instead only train them to be data-scalable. Similarly, Grace-AE does not improve the generalization of pre-trained autoencoders on unseen videos, because we train them using the pre-trained models’ training data (Vimeo-90K [55] and COCO [39]). That said, autoencoders are being constantly improved to obtain higher compression efficiency (e.g., via using more reference frames like in traditional codecs) and work better on new videos (e.g., via model adaptation).

Frame skipping: Grace rarely skips a frame, unless no packet is received at all. This is a unique benefit of data-scalable delivery. However, we acknowledge that this is not necessarily the optimal choice. We believe more user studies are needed to understand if people prefer frames decoded at a slightly lower quality (due to packet losses) or skip a few frames in hope of improving the quality of future frames.

Compute and power overhead: Grace belongs to the line of work (e.g., [27, 56]) that embraces the trend of more hardware NN accelerators. That said, Grace-AE as-is runs much more slowly than the heavily engineered codecs, so it is not applicable to low-power devices.

8 Conclusion

This paper presents Grace, a new real-time video system based on autoencoders trained to deliver video frames in a data-scalable manner (decodable with any non-empty subset of packets and graceful improvement with more packets). Though in absence of packet losses, Grace has lower coding efficiency than traditional codecs, we show that Grace allows clients to better balance video quality and frame delay.
References

[1] Aurora5 HEVC Test Results. https://www.visionular.com/en/putting-the-aurora5-hevc-encoder-to-the-test/.

[2] Features of WebRTC VR Streaming. https://flashphoner.com/features-of-webrtc-vr-streaming/.

[3] Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture. https://datatracker.ietf.org/doc/draft-ietf-tsvwg-l4s-arch/.

[4] Open Source Cloud Gaming with WebRTC. https://webrtchacks.com/open-source-cloud-gaming-with-webrtc/.

[5] OpenH264. https://github.com/cisco/openh264.

[6] QUIC FEC v1. https://docs.google.com/document/d/1Hg1SaLEI6T4rEU9j-1sovCo8VEjnuCPTClnJewj7Nk/edit.

[7] QUIC FEC v1. https://web.archive.org/web/20141103202912/http://x264dev.multimedia.cx:x80/archives/472.

[8] SI/TI calculation tools. https://github.com/VQEG/siti-tools.

[9] torchac: Fast Arithmetic Coding for PyTorch. https://github.com/fab-jul/torchac.

[10] Video Streaming Latency Report, 2020. https://www.wowza.com/wp-content/uploads/Streaming-Video-Latency-Report-Interactive-2020.pdf.

[11] Video Streaming Latency Report, 2021. https://f.hubspotusercontent20.net/hubfs/229276/2021%20Low%20Latency%20Report/streaming-video-latency-report-interactive-2021.pdf.

[12] WebRTC and IoT Applications. https://rtcweb.in/webrtc-and-iot-applications/.

[13] WebRTC Cloud Gaming: Unboxing Stadia. https://webrtc.ventures/2021/02/webrtc-cloud-gaming-unboxing-stadia/.

[14] WebRTC: Enabling Collaboration Augmented Reality App. https://arvrjourney.com/webrtc-enabling-collaboration-cebddd4c9ce06?gi=e19b1c0f65c0.

[15] WebRTC in IoT: What is the Intersection Point? https://mobidev.biz/blog/webrtc-real-time-communication-for-the-internet-of-things.

[16] Focusing on latency, not throughput, to provide a better internet experience and network quality. https://www.iab.org/wp-content/IAB-uploads/2021/09/Nokia-IAB-Measuring-Network-Quality-Improving-and-foc-p.pdf, 2021.

[17] Working latency – the next QoE frontier. https://blog.apnic.net/2021/12/02/working-latency-the-next-qoe-frontier/, 2021.

[18] FVC Workshop. https://fvc-workshop.github.io/, 2022.

[19] Google’s WebRTC implementation. https://webrtc.gs.google.com/src/, 2022.

[20] Niklas Blum, Serge Lachapelle, and Harald Alvestrand. Webrtc-realtime communication for the open web platform: What was once a way to bring audio and video to the web has expanded into more use cases we could ever imagine. *Queue*, 19(1):77–93, 2021.

[21] Eirina Bourtsoulatze, David Burth Kurka, and Deniz Gündüz. Deep joint source-channel coding for wireless image transmission. *IEEE Transactions on Cognitive Communications and Networking*, 5(3):567–579, 2019.

[22] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016.

[23] João Carreira and Andrew Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4724–4733, 2017.

[24] Guy Cote, Shahram Shirani, and Faouzi Kossentini. Optimal mode selection and synchronization for robust video communications over error-prone networks. *IEEE Journal on Selected Areas in Communications*, 18(6):952–965, 2000.

[25] Lei Dai and Liming Zhang. A joint spatiotemporal video compression based on stochastic adaptive fourier decomposition. *IEEE Signal Processing Letters*, 29:1531–1535, 2022.

[26] Linglong Dai, Ruicheng Jiao, Fumiuki Adachi, H Vincent Poor, and Lajos Hanzo. Deep learning for wireless communications: An emerging interdisciplinary paradigm. *IEEE Wireless Communications*, 27(4):133–139, 2020.
[27] Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras. Swift: Adaptive video streaming with layered neural codecs. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 103–118, Renton, WA, April 2022. USENIX Association.

[28] Tugba Erpek, Timothy J O’Shea, Yalin E Sagduyu, Yi Shi, and T Charles Clancy. Deep learning for wireless communications. In Development and Analysis of Deep Learning Architectures, pages 223–266. Springer, 2020.

[29] Silas L Fong, Salma Emara, Baochun Li, Ashish Khisti, Wai-Tian Tan, Xiaoqing Zhu, and John Apostolopoulos. Low-latency network-adaptive error control for interactive streaming. In Proceedings of the 27th ACM International Conference on Multimedia, pages 438–446, 2019.

[30] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: {Low-Latency} network video through tighter integration between a video codec and a transport protocol. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 267–282, 2018.

[31] Deniz Gündüz, Paul de Kerret, Nicholas D Sidiropoulos, David Gesbert, Chandra R Murthy, and Mihaela van der Schaar. Machine learning in the air. IEEE Journal on Selected Areas in Communications, 37(10):2184–2199, 2019.

[32] Stefan Holmer, Mikhail Shemer, and Marco Paniconi. Handling packet loss in webRTC. In 2013 IEEE International Conference on Image Processing, pages 1860–1864. IEEE, 2013.

[33] ITU-T Recommendation. Subjective video quality assessment methods for multimedia applications. 1999.

[34] Bart Jansen, Timothy Goodwin, Varun Gupta, Fernando Kuipers, and Gil Zussman. Performance evaluation of webRTC-based video conferencing. ACM SIGMETRICS Performance Evaluation Review, 45(3):56–68, 2018.

[35] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings, 2014.

[36] David Burth Kurka and Deniz Gündüz. Deepjscc-f: Deep joint source-channel coding of images with feedback. IEEE Journal on Selected Areas in Information Theory, 1(1):178–193, 2020.

[37] Hoang Le, Liang Zhang, Amir Said, Guillaume Sautiere, Yang Yang, Pranav Shrestha, Fei Yin, Reza Pourrea, and Auke Wiggers. Mobilecodec: neural inter-frame video compression on mobile devices. In Proceedings of the 13th ACM Multimedia Systems Conference, pages 324–330, 2022.

[38] Yang Li, Guangcan Liu, Yubao Sun, Qingshan Liu, and Shengyong Chen. 3d tensor auto-encoder with application to video compression. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 17(2):1–18, 2021.

[39] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In European conference on computer vision, pages 740–755. Springer, 2014.

[40] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. Dvc: An end-to-end deep video compression framework. In 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pages 10998–11007, 2019.

[41] David JC MacKay. Fountain codes. IEEE Proceedings-Communications, 152(6):1062–1068, 2005.

[42] David JC MacKay and Radford M Neal. Near shannon limit performance of low density parity check codes. Electronics letters, 33(6):457–458, 1997.

[43] Kyle MacMillan, Tarun Mangla, James Saxon, and Nick Feamster. Measuring the performance and network utilization of popular video conferencing applications. In Proceedings of the 21st ACM Internet Measurement Conference, pages 229–244, 2021.

[44] Alexandre Mercat, Marko Viitanen, and Jarno Vanne. Uvg dataset: 50/120fps 4k sequences for video codec analysis and development. In Proceedings of the 11th ACM Multimedia Systems Conference, pages 297–302, 2020.

[45] Mohamed Moulay and Vincenzo Mancuso. Experimental performance evaluation of webRTC video services over mobile networks. In IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pages 541–546. IEEE, 2018.

[46] Marcin Nagy, Varun Singh, Jörg Ott, and Lars Eggert. Congestion control using fec for conversational multimedia communication. In Proceedings of the 5th ACM Multimedia Systems Conference, pages 191–202, 2014.

[47] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and
A Details of autoencoder architecture and training

A.1 I-frame compression model

Figure 16: High-level architecture of I-frame compression autoencoder.

Autoencoder architecture for compressing I-frames consists of an encoder, a decoder, a hyper-encoder, and a hyper-decoder networks.

Encoder network is used to generate a latent representation of the input image. It consists of five blocks and a fully connected layer in the end. Each block consists of a convolutional layer, generalized divisive normalization layer, and a Spatial Feature Transform Layer. The last fully connected layer serves as a converter resized for the latent representation size. It can be easily fine-tuned by freezing the rest of the layers and re-training only the fully connected layer for a short time to fit the desired size for the latent representation of the image.

Hyper-encoder and hyper-decoder networks are used to generate side-information from the latent representation of the image. The output of the hyper-encoder is used as an input to hyper decoder to generate the parameters (mu, sigma) for the Gaussian entropy model, which approximates the distribution of the latent representation of the image. The hyper-encoder
consists of three blocks, where each block consists of a convolutional layer, Spatial Feature Transform layer, and a LReLU. The hyper-decoder consists of three blocks, each of which consists of a convolutional layer and a LReLU.

Decoder uses the reconstructed latent image representation and the reconstructed side-information generated by the hyper-encoder to reconstruct the image. Decoder consists of five blocks, each of which consists of convolutional layer, a generalized divisive normalization layer, and a Spatial Feature Transform layer.

We apply quantization to both latent image representation and side-information generated by hyper-encoder before they make it into the decoder and hyper-decoder respectively.

Autoencoder model to compress I-frames was trained on the COCO Dataset consisting of COCO 2014 and COCO 2017 which sum up to 201,000 images in total. We used the batch size of 8, with learning rate of 0.0001. It took 1,980,000 iterations to train the base model with 0% packet loss, and 100,000 iterations to fine-tune the model with packet loss.

A.2 P-frame compression model
DVC model to compress P-frames mimics the mpeg encoder and decoder structure, with residual, motion vector encoder and decoder, and motion compensation substituted with convolutional networks.

Motion estimation is realized in three steps. First, we use optical flow estimation using a library to estimate motion vectors from optical between two frames. Then, optical flow is encoded using a motion vector encoder. The motion vector encoder consists of four convolutional layers, separated by three generalized divisive normalization layers. After the encoder, we apply quantization and loss to the motion vector data, which then is fed into the decoder. The motion vector decoder consists of four deconvolution layers, with three inverse generalized divisive normalization layers.

We take the output of the motion vector decoder and output of residual decoder and feed it into the motion compensation network. Motion compensation network first takes in the reconstructed motion vector and a reference frame, where reference frame is warped using the reconstructed motion vector. Then, the resulting warped frame together with the reference frame and reconstructed motion vector are fed into a convolutional network, which consists of six convolutional layers with average pooling steps in between them. The output of this convolutional network is the reconstructed frame.

DVC model to compress P-frames was trained on the 90k Vimeo Dataset, with batch size of 4, learning rate of 0.0001 and learning rate decay of 0.1. To control bitrate, we scaled the PSNR component in the loss term by a constant factor $\alpha$, making it larger in proportion to BPP. Furthermore, for the purposes of saving memory that models occupy and training speed, we chose model with $\alpha = 1024$ as the "backbone" and fine-tuned only 10%-30% of layers to adjust to other bitrates. This means that all models share 90%-70% of the layers, with only the last one to two convolutional layers of the residual encoder, last one to two convolutional layers of the motion vector encoder, first one to two convolutional layers of residual decoder and first one to two convolutional layers of the motion vector decoder being trained. This allowed us to save over $2-3 \times$ in model storage memory, as well as for each bitrate model to converge within 10,000 iterations with the application of packet loss.

A.3 Distribution of video content complexity

![Figure 17: Spatial information (SI) and temporal information (TI) of test videos](image)

To validate the test videos that we use cover different content complexities and movements, we calculate the spatiotemporal complexity of the video. We use Spatial Information (SI) and Temporal Information (TI) [33], which are frequently-used metrics to measure the spatiotemporal complexity and a larger SI/TI means that the video has a higher spatiotemporal complexity. The metrics are calculated by the tool [8] provided by Video Quality Experts Group (VQEG) and the result is shown in Figure 17.

The result validates that (i) the spatiotemporal complexity of the videos we used covers a wide range: SI is ranging from 15 to 85 and TI is ranging from 3 to 25. (ii) Our test videos covers all the following types: high spatial complexity and high temporal complexity, high spatial complexity but low temporal complexity, low spatial complexity but high temporal complexity, and low spatial complexity and low temporal complexity.

A.4 Visualization of Grace-AE’s Decoded Frames

Figure 18 gives a concrete visualization of how well Grace-AE performs under packet losses. When there is no loss, both the pretrained autoencoder and Grace-AE can reconstruct the frame with a decent quality (Figure 18b) with 0.9952 and 0.9943 SSIM, respectively. When 50% loss are applied to the last 3 consecutive frames, the pretrained autoencoder
Figure 18: Visualization of the decoded image under different conditions. (a) is the original frame. (b) is the reconstructed frames using the pretrained autoencoder when there is no loss. (c)-(d) are the reconstructed frames using the pretrained model and Grace-AE when 50% loss happens in 3 consecutive frames, respectively.

model fails to reconstruct the original frame with a good quality: annoying distortions exists in the decoded frame (at the bottom of the glass container) and SSIM decreased to 0.9630. However, under the same loss, Grace-AE can have a much higher SSIM (0.9830) and the distortions are much less visible (Figure 18d) comparing to the pretrained model.