# Continuous Prefetch for Interactive Data Applications

Haneen Mohammed
DSI, Columbia University
ham2156@columbia.edu

Eugene Wu
DSI, Columbia University
ewu@cs.columbia.edu

Ziyun Wei
Cornell University
zw555@cornell.edu

Ravi Netravali
UCLA
ravi@cs.ucla.edu

## ABSTRACT

Interactive data visualization and exploration (DVE) applications are often network-bottlenecked due to bursty request patterns, large response sizes, and heterogeneous deployments over a range of networks and devices. This makes it difficult to ensure consistently low response times ($< 100$ms). KHAMELEON is a framework for DVE applications that uses a novel combination of prefetching and response tuning to dynamically trade-off response quality for low latency.

KHAMELEON exploits DVE's approximation tolerance: immediate lower-quality responses are preferable to waiting for complete results. To this end, KHAMELEON progressively encodes responses, and runs a server-side scheduler that proactively streams portions of responses using available bandwidth to maximize user-perceived interactivity. The scheduler involves a complex optimization based on available resources, predicted user interactions, and response quality levels; yet, decisions must also be made in real-time. To overcome this, KHAMELEON uses a fast greedy heuristic that closely approximates the optimal approach. Using image exploration and visualization applications with real user interaction traces, we show that across a wide range of network and client resource conditions, Khameleon outperforms existing prefetching approaches that benefit from *perfect* prediction models: KHAMELEON always lowers response latencies (typically by 2–3 orders of magnitude) while keeping response quality within 50–80%.

## 1. INTRODUCTION

Interactive data visualization and exploration (DVE) applications, such as those in Figure 1, are increasingly popular and used across sectors including art galleries [13], earth science [38], medicine [9], finance [45], and security [27]. Like
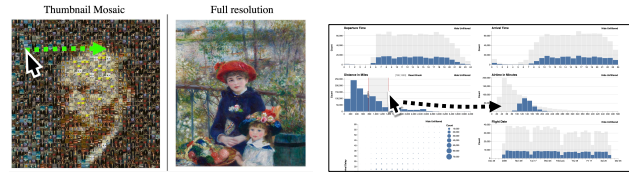
Figure 1: Two DVE applications. (a) Image exploration: hover over the left thumbnails mosaic to load full image on the right. (b) Falcon visualization [49]: drag and resize range filters in a subset of charts to update the others. We use these in our experiments §6.

typical web services, DVE applications may be run on heterogeneous client devices and networks, with users expecting fast response times under 100 ms [16, 41, 60]. However, the resource demands of DVE applications are considerably magnified and highly unpredictable, making it difficult to achieve such interactivity.

Traditional interactive applications are based on point-and-click interfaces such as forms or buttons, where there may be seconds or minutes of delay between user requests. In contrast, DVE applications update the visual interface continuously as the user drags, hovers, or otherwise manipulates the interface [18] (Figure 1). For example, all of the charts in Figure 1b are updated continuously as the user drags and resizes range filters. In short, DVE applications support a large number of potential requests rather than a few buttons, exhibit **bursty** [7] user interactions that generate a huge number of back-to-back requests with nearly **no "think time"** between them, and issue **data-dense** requests for tens of kilobytes to megabytes of data in order to render detailed statistics or high-resolution images [14].

As a result of these combined factors, DVE applications place considerable and unpredictable pressure on both network and server-side data processing resources. Fortunately, the database and visualization communities have made considerable progress in reducing server-side [45, 83, 42, 12, 78] and client-side [27, 42] data processing and rendering latencies. However, *network bottlenecks still persist*, and can cripple user-facing responsiveness even if server and client-side overheads are eliminated. Addressing network bottlenecks is becoming paramount with the continued shift towards cloud-based DVE applications that must remain responsive across a wide variety of client network conditions (e.g., wireless, throttled).

The primary approach to masking network delays in interactive applications is prefetching [21, 5, 36, 20, 75, 11, 2], where responses for predicted future requests are proactively cached on the client. Prefetching benefits inherently depend on prediction accuracies, but sufficiently high accuracies have remained elusive, even for well-studied classic applications such as web pages [63]. DVE applications pose an even more challenging setting for several reasons. Their bursty request patterns, when combined with data-dense responses, can easily exceed the bandwidth capacities of existing networks and cause persistent congestion. At the same time, the massive number of potential requests makes building a near-perfect predictor that can operate over long time horizons infeasible—developing such oracles is an open problem. Thus, prefetching for DVE applications is either ineffective or wastes precious network resources which, in turn, can cause (detrimental) *cascading slowdowns* on later user requests.

In this paper, we depart from traditional prefetching frameworks that hope to accurately predict a small number of future requests to prefetch, towards a framework that continuously and aggressively hedges across a large set of potential future requests. Such framework should also decouple request burstiness from resource utilization so that the network does not get overwhelmed at unpredictable intervals, and instead can consistently employ all available network resources for subsequent prefetching decisions.

A trivial, but undesirable, way to meet these goals is to limit the user's ability to interact with the interface, thereby reducing the burstiness and scope of possible requests. Instead, we leverage the fact that DVE applications are **approximation tolerant**, meaning that response quality can be dynamically tuned [44, 82, 25] to enable more hedging within the available resources (at the expense of lower response quality). Of course, this introduces a fundamental tradeoff: the prefetching system can focus on low-quality responses for many requests to ensure immediate responses, or high-quality responses for a few requests at the risk of more cache misses and slow responses. Balancing this tradeoff requires a *joint optimization between response tuning and prefetching*, which, to date, have only been studied independently. This involves a novel and challenging scheduling problem, as the optimization space needs to consider the likelihood of the user's future access patterns over a large number of possible requests, applications preferences between response quality levels and responsiveness, and limited resource conditions. At the same time, the scheduler must run in real-time.

We present **Khameleon**, a novel prefetching framework for DVE applications that are bottlenecked by request latency and network transfer. KHAMELEON dynamically trades off response quality for low latency by leveraging two mechanisms that collectively overcome the joint optimization challenges.

First, we leverage progressive encoding[1] to enable fine-grained scheduling. Each response is encoded as an ordered list of small blocks such that any prefix is sufficient to render a lower quality response, and additional blocks improve response quality. This encoding is natural for DVE applications, which can leverage existing encodings for e.g.,

images [72, 25] and visualization data [5]. Progressive encoding lets the prefetching system vary the prefix for a given response based on its (predicted) future utility to the user.

Second, we shield network and server resources from prediction errors and client burstiness by using a push-based model, rather than having clients issue requests directly to the server. The server streams blocks for likely requests to the client cache using the available (or user-configured) network capacity. The server-side scheduler determines a global sequence of blocks to continually push to the client. By default, it assumes that all requests are equally likely. However, the application can define a predictor to estimate future requests; in this case, the client uses the predictor to periodically send forecasted probability distributions to the scheduler, which updates the global sequence accordingly. To ensure real-time operation, the scheduler uses a greedy heuristic, which closely approximates the optimal algorithm.

KHAMELEON is a framework that is compatible with existing DVE applications. KHAMELEON transparently manages the request-oriented communication between the DVE client and server, and shields developers from the challenges of the joint optimization problem. Developers can instead focus on high-level policies, such as determining their preference between latency and quality, and developing application-specific progressive encoding schemes and prediction models. §3.4 describes the steps that a developer would take to use KHAMELEON with an existing DVE application.

We evaluate KHAMELEON using the two representative DVE applications in Figure 1. Our experiments consider a broad range of network and client resource conditions, and use real user-generated interaction traces. Across these conditions, we find that KHAMELEON is able to avoid network congestion and degraded user-facing responsiveness that arises from using indiscriminate prefetching (even if that prefetching uses a 100% accurate predictor). For instance, for the image exploration application, KHAMELEON (using a simple predictor [74]) reduces response latencies by up to 3 orders of magnitude ($> 10$s to $\approx 10$ms) and maintains a response quality of 50–80%. Similarly, with the Falcon data visualization [49], KHAMELEON's progressive encoding improves response latencies on average by $4\times$ and improves response quality by up to $1.6\times$. Our experimental setup also reveals that porting existing applications to use KHAMELEON entails minimal burden. For example, modifying Falcon to use KHAMELEON as the communication and prefetching layer required fewer than 100 lines of code to issue requests to the KHAMELEON client library and use a formal predictor.

Our contributions include 1) design and implementation of KHAMELEON, a framework that combines real-time prediction, progressive encoding, and server-side scheduling for interactive DVE applications, 2) formalization and fast greedy implementation of the server-side scheduling optimization problem that balances request qualities and likelihoods, 3) and extensive evaluation using two DVE applications to highlight the benefits of the KHAMELEON design.

## 2. DVE APPLICATIONS

Cloud-based DVE applications are *information dense*, in that they render hundreds or thousands of data items (e.g., records, images) that users can directly interact with. The request patterns from these interactions are ***bursty*** with

---

[1]Progressive encoding is distinct from progressive computation, such as online aggregation [32], which returns full, yet approximate, responses by processing a sample of the database. §7 discusses how progressive computation can exacerbate network bottlenecks in more detail.

**negligible think time** between requests. These characteristics lead to a rate of requests that stresses the network, often exceeding the available capacity and resulting in congestion.

In order to address the potential network bottlenecks, KHAMELEON leverages two key properties of DVE applications. First, interactions are **preemptive**: since responses can arrive out of order (e.g., due to network or server delays), the client renders the data for the most recent request and (silently) drops responses from older requests to avoid confusing the user [79, 80]. Second, they are **approximation tolerant**: it is preferable to quickly render a low-quality response (e.g., fewer points [61] or coarser bins [42]) than to wait for a full quality response. As concrete examples, consider the following two DVE applications which exhibit these properties; we use both in our evaluation (§6).

**Large-scale image exploration.** We developed an image exploration application (Figure 1a) based on large image datasets of e.g., art [13], satellite data [38], cellular microscopy [9], and maps [30]. The user's mouse can hover one of $10,000$ image thumbnails on the left (akin to a zoomed-out map view) to view the full resolution $1.3 - 2Mb$ on the right (akin to a zoomed-in map tile).

We consider this an exemplar and difficult DVE application because it has a high request rate, large response sizes, and with 10K thumbnails, it is difficult to build an accurate predictor for. For instance, from the user traces used in our experiments, clients request up to 32 images per second $(32–64$ MB)/s),[2] not including any prefetching requests. In addition, this application imposes fewer interaction restrictions than existing exploration applications that are plagued by prefetching inefficiencies. For instance, applications like Google Maps only let users pan to adjacent tiles and incrementally zoom; this both simplifies prediction and limits the rate of images that the user can request at a time.

**Interactive data visualizations.** Falcon [49] is a state-of-the-art interactive visualization application specifically optimized for prefetching (Figure 1b). As the user selects and resizes range selections in any of the charts, the other non-selected charts immediately update their statistics to reflect the conjunction of all of the selections. The challenge is that the space of possible combinations of selections is exponential in the number of charts, and is infeasible to fully precompute and send to the client up front. Yet, even movements across a single pixel trigger many requests.

In order to minimize interaction delays, the Falcon developers [49] manually implemented prefetching to mask request latencies. They observed that the user can only interact with one chart at a time, and selections in the other charts are fixed. When the user's mouse moves onto chart A, Falcon sends SQL queries to a backend database to compute low dimensional data cube slices between chart A and each of the other charts to update. Once these slices are constructed, user interactions in chart A are handled instantaneously.

Falcon's *predictor* prefetches data slices when the user hovers over a chart, and it *progressively encodes* the data slices as cumulative counts. However, these policies are hardcoded in a monolithic codebase, making it challenging to improve the predictor (e.g., estimate the chart the mouse will interact with, rather than wait for a hover event), response encoding (e.g., pixel-resolution and a coarse resolution ), or user

---

[2]For reference, streaming HD and 4K video typically requires 5-20 Megabits (Mb)/s.
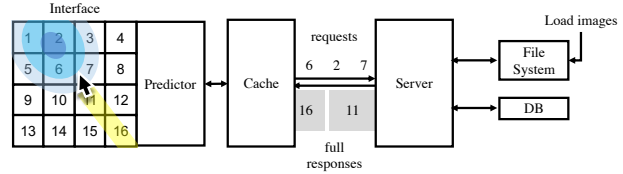


(a) Traditional prefetching architecture
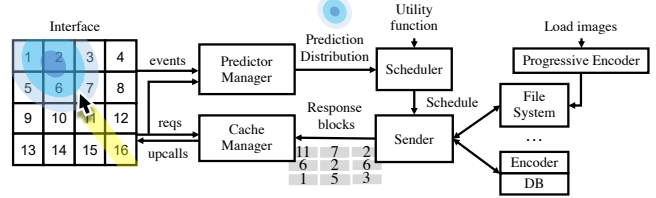


(b) Khameleon architecture

Figure 2: Comparing KHAMELEON to a traditional prefetching architecture for an image exploration DVE application. The interface is a $4 \times 4$ grid of image thumbnails. The cursor moved over images 16 and 11 (yellow line) and is now positioned over image 7; the probability of the mouse's future position is a gaussian distribution illustrated by the blue ellipses (center is the highest probability). The gray boxes are sized according to response sizes. KHAMELEON separates the predictor from the cache, sends probability distributions instead of explicit requests, and uses a scheduler to determine the sequence of small request blocks to send to the client.

preferences (e.g., which attributes they favor). §6.4 describes the details of how we adapted Falcon to use KHAMELEON as the communication layer, and switched its database from OmniSci to PostgreSQL.

## 3. KHAMELEON OVERVIEW

This section first describes traditional prefetching architectures and their limitations for DVE applications. It then provides a high-level design overview of KHAMELEON, explaining its individual components (we elaborate on each in the following sections), how they collectively overcome the limitations of prior architectures, and how existing DVE applications can be seamlessly migrated to use KHAMELEON.

### 3.1 Traditional Prefetching Architecture

Figure 2(a) depicts the workflow of a common prefetching architecture for an image exploration DVE application. In this application, a user interacts with a grid composed of 16 image thumbnails such that mousing over an image enlarges its display size. As the user moves the mouse, the local cache manager receives requests and immediately responds if the data is cached, or forwards the request to the server. In parallel, the gaussian distribution representing predictions of the mouse's future location is updated based on the mouse's current position (to improve prediction accuracy), and is used to pick a set of requests to prefetch; these requests are issued to the cache manager in the same way.

In this example, the user mouse has moved quickly along the yellow path, triggering requests for images 16 and 11. Given the bursty nature of these requests, the corresponding responses are still being transmitted as the next user-generated request is issued (for image 7).

Unfortunately, because the full response data for images 16 and 11 fully utilize the available network bandwidth, the response for 7 will have to contend with those old responses, delaying its arrival. To make matters worse, the client prefetching mechanism will concurrently issue requests for the k most likely next requests (2 and 6 in this example).

**Limitations.** The problem here is that the information that enables the most accurate predictions (i.e., the mouse's current position) is available exactly when the user issues bursts of requests that already congest the network. This has two consequences. First, this largely eliminates any prefetching benefits, and highlights its drawbacks: accurate prefetching requests are unlikely to return to the cache before the user explicitly requests them (recall that DVE applications experience low user think times), and inaccurate prefetching requests add unnecessary network congestion that slows down explicit user-generated requests and future prefetching. Second, it is difficult to know what requests should be prefetched during the times between user interactions because the user, by definition, is not generating events; unfortunately, prefetching everything is impractical given the high data footprint of DVE applications (§2).

## 3.2   Khameleon Architecture

KHAMELEON (Figure 2(b)) consists of client-side and server-side libraries that a cloud-based DVE application can import and use to manage data-intensive network communication. These components operate as follows to overcome the aforementioned limitations of traditional prefetching architectures.

The client-side library serves to decouple prefetching requests from the (bursty) network utilization triggered explicitly by the user. User-generated requests are not sent out on the network, and instead are registered with the local *Cache Manager*. The Cache Manager waits until there is cached data to fulfill the request, and then makes an application upcall to update the interface with that data. This approach helps absorb high user request rates. As this happens, client events (e.g., mouse movements) and requests are also passed to an application-provided *Predictor Manager* that continually updates a distribution of predicted future requests and sends a summary of that distribution (e.g., the parameters of a gaussian distribution) to the server.

The server-side library uses intelligent push-based scheduling and progressive encoding of responses to make the most of the available network resources, i.e., balancing user-perceived latency and response utility while hedging across potential future requests. The *Scheduler* continually maintains a schedule of response blocks to push to the client; the set of blocks covers a wide range of explicit and anticipated requests, e.g., images 11, 7, etc. in Figure 2(b). The specific sequence of blocks depends on the predicted request probabilities received from the client, as well as an optional application-provided *Utility Function* that quantifies the "quality" of a response based on the number of prefix blocks available. Note that a single block is a complete response, with additional blocks improving "quality" according to the Utility Function. A separate *Sender* thread reads the schedule and retrieves blocks from backend systems. For example, the file system could be pre-loaded with the blocks for progressively encoded images, or a database could dynamically execute queries and progressively encode the results before returning the subset of required blocks to the Sender.

Finally, the server streams the sequence of response blocks to the client, which updates its local cache accordingly.

As we describe in §3.4, this design enables the application to independently improve its prediction model, utility functions, data encodings, backends, or scheduling policies. The KHAMELEON architecture is agnostic to how the application client interprets and decodes the blocks, as well as to the specific backend system that is employed.

## 3.3   System Components

**Predictor Manager.** This client-side component relies on an application-provided predictor to forecast future requests (as a probability distribution over the possible requests), and periodically sends those predictions to the server. Predictors must satisfy two properties. First, at time t, the predictor must return a probability distribution $P^t(q|D)$ over requests q and future times $t + D$. Second, it must be *Anytime*, so that the Predictor Manager can ask for distributions of predicted requests to send the server at any time during system operation. It is also important that the predictor's state is efficient to maintain, and that the distributions can be compactly represented for transmission to the server. These mechanisms enable the Predictor Manager to control policies for how often to make and send distributions.

**Progressive Results and Caching.** Each request's progressively encoded response is modeled as an ordered list of fixed size blocks; any prefix is sufficient to render (a possibly lower quality) result, and the full set of blocks renders the complete result. Smaller blocks can be padded if block sizes differ. Our client-side cache implementation uses a ring buffer (FIFO replacement policy) for its simplicity and determinism; in particular, this simplifies the server-side scheduler's ability to track cached state at the client, since the FIFO policy can be simulated without explicit coordination.[3]

During operation, the cache puts the $i^{th}$ block received from the server into slot i%C, where C is the cache size. The cache responds to a request if there is $\geq 1$ response block in the cache for the request. To implement preemptive interactions (§2), the cache assigns each request an increasing logical timestamp when it is registered, and deregisters all earlier timestamps when an upcall for request i is made.

**Utility Functions.** In practice, the first few blocks of a response are likely to contribute more than subsequent blocks [61, 72, 43]. To quantify this, the application can optionally provide a monotonically increasing *Utility Function* $U : [0, 1] \mapsto [0, 1]$, which maps the percentage of data blocks for a request to a utility score. A score of 0 means most dissimilar, and 1 means identical to the full result, in expectation. By default, KHAMELEON conservatively assumes a linear utility function. As an example, Figure 3 plots the utility curve for the image exploration application, which is based on the average visual structural similarity measure [73] between the progressive result and the full image.

**Scheduler and Backends.** The Scheduler decides the sequence of blocks to push to the client. It schedules in batches of C blocks (the client cache size) because the client's ring buffer will overwrite itself once it is filled. A given batch (a *schedule*) is chosen to maximize the user's expected utility with respect to the probability distribution over future requests. The separate *Sender* thread reads the schedule to

---

[3]Other deterministic replacement policies are possible and incorporating them is left for future work.
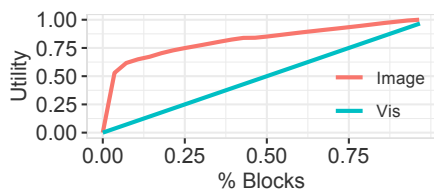
Figure 3: Utility functions: image exploration application uses structural similarity (red), visualization application uses system default linear function (blue).

learn about the order in which blocks should be retrieved from the backend and placed onto the network. The backend may be a file system, a database engine, a connection pool, or any service that can process requests and return progressively encoded blocks. Note that, given a progressive encoder, any backend can be retrofitted by encoding its results. We retrofit PostgreSQL for our visualization experiments.

By default, we assume that retrieving blocks from the backend incurs a predictable delay. In addition, we assume that the backend is *scalable*, in that the delay does not increase considerably when more concurrent queries are issued (e.g., speculatively for prefetching). This is natural for pre-computed responses or backends such as a file system or key value store. In cases where the backend can only scale to a limited number of requests, KHAMELEON employs a heuristic to limit the amount of speculation in accordance with the supported scalability (§5.4).

## 3.4 Adapting Applications to Khameleon

This subsection describes how a DVE application (image exploration in this case) can be easily and incrementally adapted to use KHAMELEON. Recall that the application issues an image request when the user's mouse hovers over a thumbnail; the server retrieves the full-sized image from the file system, and sends it back to the client.

To use KHAMELEON, the application should provide a progressive encoding of its responses, a utility function, and a predictor. Since traditional requests and responses are special cases of KHAMELEON's predictor and encoder, we start with generic defaults for these components. The generic encoder treats each image as a response with a single block, and the predictor treats each request as a point distribution. By specifying this, an immediate benefit is that the scheduler will use the point distributions to select the full requested image (as in the existing application), and use the remaining bandwidth to push random images for the client to cache.

We now show how a developer Jude can improve each component for her application. A benefit of KHAMELEON's modular design is that the components can be improved independently.

**Improve the Encoder:** Finer-grained blocks help improve the scheduler's ability to hedge across many requests given finite bandwidth resources. Since JPEG is a progressive encoding, Jude replaces the naive encoder with a JPEG encoder and configures the system with the block size. Further, she can adjust the JPEG encoding parameters to create finer-grained block sizes, or switch to an alternative progressive encoding altogether [57].

**Improve the Utility Function:** By default, KHAMELEON uses the linear utility function, where each block contributes the same additional utility to the user. Jude computes the structural similarity [73] for different prefix sizes over a sample of images, and uses this to derive a new utility function (e.g., Figure 3).

**Improve the Predictor:** Jude now uses her application expertise to incrementally improve the predictor. One direction is to weigh the point distribution with a prior based on historical image access frequency. Alternatively, she could directly estimate the user's mouse position using a variety of existing approaches [56, 76, 77, 29, 4]. She can assess the benefits of any modifications to the predictor based on its empirical accuracy over live or historical user traces, or higher-level metrics such as cache hit rates and number of blocks available for each request—KHAMELEON reports both.

In §6.4, we describe how we adapted the state-of-the-art Falcon DVE application to KHAMELEON with < 100 LOC.

## 4. PREDICTOR MANAGER

The application-provided prediction model $P^t(q|D, e_t)$ uses interaction events and/or requests $e_t$ up until the current time t in order to estimate the probability of request q at D time steps in the future. Of course, there exist a wide range of prediction models that satisfy this definition, with the appropriate one varying on a per-application basis. For example, button and click-based interfaces benefit from Markov-based models [31, 8, 17], whereas continuous interactions such as mouse- or hover-based applications benefit from continuous estimation models [4, 74, 56, 76]. Regardless of the prediction model used, a commonality with respect to KHAMELEON is that the events $e_t$ (e.g., mouse movements, list of previous user actions) are generated on the client, whereas the predictions are used by the server-side scheduler.

Given these properties, KHAMELEON provides a generic API for applications to register their desired predictors; KHAMELEON is agnostic to the specific prediction model being suggested. The API (described below) decomposes a predictor into client-side and server-side components, and KHAMELEON's Predictor Manager handles the frequency of communication between the two components. The main requirement is that the predictor is usable at any time to estimate a probability distribution over possible requests at arbitrary time steps. We note that KHAMELEON does not mandate a specific prediction accuracy. However, KHAMELEON can report prediction accuracies, as well as application-level performance metrics resulting from those accuracies, based on live and historical user traces; developers can then use this feedback to continually improve their predictors.

**Predictor decomposition.** Applications specify the predictor $P^t$ as server and client components (correspondingly colored):

$$P^t(q|D, e_t) = P_s^t(q|D, s_t)P_c^t(s_t|D, e_t)$$

The client component $P_c^t$ collects user interaction events and requests $e_t$ and translates this information into a byte array that represents the predictor state $s_t$. $s_t$ may be the most recent request(s), model parameters, the most recent user events, or simply the predicted probabilities themselves. The server uses $s_t$ as input to $P_s^t$ in order to return future request probabilities for the KHAMELEON scheduler's joint optimization between prefetching and response tuning.

Importantly, this decomposition is highly flexible and can support a variety of different configurations for predictor components. For example, a pre-trained Markov model [31,

2301

8, 17] may be instantiated on the server as $P_s^t$, and the client may simply send each event to the server ($s_t = e_t$). Alternatively, the Markov model could be placed on the client as $P_c^t$, with the state sent being a list of the top k most likely requests, and the server component assuming that all non-top k requests have probability of $\approx 0\%$.

**Devising A Custom Predictor.** We now walk through the design of a custom predictor for interfaces with static layouts, i.e., the two example DVE applications in Figure 1. These are the predictors that we use in our experiments in §6. We note that the purpose here is to elucidate the operation of an anytime predictor and the process that a developer may follow in designing a suitable (custom) one for their application. We do not claim that the following predictor is the best possible one for a given application.

The two DVE applications in Figure 1 both use a fixed set of static layouts: one uses a grid of thumbnails, while the other uses a set of fixed-size charts. Since requests are only generated when a user's mouse is positioned atop a widget, the mouse position $(x, y)$ is a rich signal for predicting future requests. More specifically, the bounding boxes of all widgets in the current layout l, denoted $P_l$, can directly translate a distribution of mouse locations $P_s^t(x, y|D, s_t)$ into a distribution over requests:

$$P^t(q|D, e_t) = P_l(q|D, x, y, l)P_s^t(x, y|D, s_t)P_c^t(s_t, l|D, e_t)$$

We model $P_s^t(x, y|D, s_t)$ as a gaussian distribution represented by the centroid and a $2 \times 2$ covariance matrix—this state is sent to the server. We choose a fixed set of D values $(50, 150, 250, 500ms$ in our experiments) to predict over, and linearly interpolate between these times. Thus, the state $s_t$ only consists of 6 floating point values for each D, which we estimate using a naive Kalman Filter [74] on the client, and decode into a request distribution on the server.

Creating a custom predictor takes careful thought, however note that *any* cloud application that uses prefetching needs to develop or adapt a predictor. Our experiments in §6.3 show that KHAMELEON is effective even with an imperfect predictor such as the Kalman Filter, because our novel scheduler explicitly and robustly account for predictor uncertainty in its joint scheduling problem.

# 5. SCHEDULER

KHAMELEON's server-side scheduler takes as input a utility function U and a probability distribution over future requests, and allocates finite network bandwidth and client cache resources across progressively encoded data blocks to maximize the expected user utility. Ultimately, it balances competing objectives: ensuring high utility for high probability requests and hedging for lower probability requests (i.e., sending some blocks for a low-quality response).

There are several challenges. First, the scheduler must track previously sent blocks and ensure they are not evicted from the client's circular cache by the time they are needed. Second, the scheduler needs to be real-time in order to not block data transmission, but still must adjust its scheduling decisions quickly when new predictions arrive from the client. This section presents the formal scheduling problem description, an ILP-based solution, and a greedy approximation.

## 5.1 Problem Definition

Let time be discretized such that each time interval $[t, t+1)$ is the time that it takes for the server to add one block of
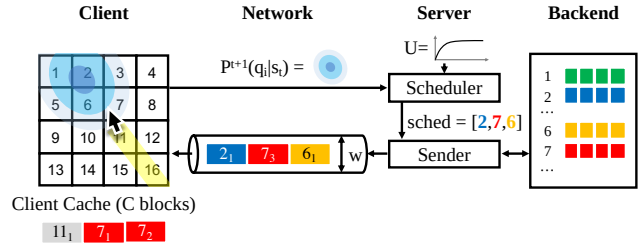


Figure 4: Setting for KHAMELEON's scheduling problem.

a response onto the network. In this problem definition, we assume that each response is progressively encoded into $N_b$ equal-sized blocks.

Let $Q = q_1, \ldots, q_n$ be the set of all possible requests. In Figure 4, there are $n = 16$ possible image requests with ids 1 to 16. The gaussian parameters estimated at time t are the state $s_t$. The scheduler has received predictor state $s_t$, which lets it estimate the probability $P(q_i|D, s_t)$ of $q_i$ being issued at D time steps in the future. Let us assume that at the start of scheduling, $t = 0$.

The client cache can hold C blocks, and the network bandwidth is w blocks per time interval. The cache at time t contains $B_i^t$ blocks for $q_i$. In the example, $2_1$ refers to the first block in image 2. The cache holds the first block of image 11 ($B_{11}^t = 1$), and the first two blocks of 7 ($B_7^t = 2$). Thus, $B^{t+1} = \{B_1^{t+1}, \ldots, B_n^{t+1}\}$ is the allocation at the end of the interval $[t, t+1]$.

PROBLEM 1 (SERVER-SIDE SCHEDULING). *Find the best next allocation* $B^{t+1}$ *that maximizes* $V(B^t)$, *given the cache* $B^t$ *and predictor state* $s_t$:

$$V(B^{t+D}) =$$
$$\max_{B^{t+D+1}} \left\{ \sum_i U(B_i^{t+D+1})P(q_i|D+1, s_t) + gV(B^{t+D+1}) \right\} \quad (1)$$

Our objective function V includes two terms (colored in formula and text). The first term is the expected user utility at the next timestep $t + D + 1$. It weighs the utility of $B_i^{t+d+1}$ blocks (using the utility function) for request i by its probability. The second term recursively computes the future benefits. This enforces the dependency between time intervals—it accounts for the long term and steers the scheduler towards a global optimum. $g \in [0, 1]$ is a discount on the future. $g = 0$ means we only care about the next timestep, and $g = 1$ means we care about all timesteps equally.

In Figure 4, the scheduler computes the best allocation for the next three time steps as the requests $2, 7, 6$. The client cache's deterministic replacement policy lets the sender push the appropriate block sequence $2_1, 7_3$, then $6_1$.

## 5.2 Approximate ILP Scheduler

Equation 1 is intractable because it contains an infinite recursion, as well as terms that are unknown at prediction time t. However, due to the design of the client cache as a circular buffer, the cache will overwrite itself after every C blocks. Thus, we approximate the solution by optimizing over a finite horizon of C blocks:

$$V(s_t, B^t) = \max_{B^{t+1}, \ldots, B^{t+C}} \sum_{k=1}^{C} \left( g^{k-1} \sum_{i=1}^{n} U(B_i^{t+k})P(q_i, k) \right) \quad (2)$$

This formulation is a Markov Decision Process [59], where actions (chosen block) in each state (contents of the cache) receive a reward (user utility). We now describe an ILP-based scheduler, followed by a fast real-time approximation. In §8, we discuss the relationship with reinforcement learning and future extensions.

**Objective Function.** Equation 2 can be expressed as an integer linear programming (ILP) problem. ILP problems require a linear objective, but the utility function U could be arbitrarily concave. We linearize it by approximating U with a step function $\tilde{U}$, defined such that $\tilde{U}(0) = 0$ and $\tilde{U}(b) = \sum_{i=1}^{b} g(i)$ where:

$$g(i) = U\left(\frac{i}{N_b}\right) - U\left(\frac{i-1}{N_b}\right) \mid i \in [1, N_b]$$

This approximation has no impact on the final result because U is already discrete due to discrete block sizes.

Let $U_{i,j}^t$ denote the expected utility gain of the j-th block for $q_i$ sent during time interval $[t-1, t]$, where $t \in [1, C]$. Because this block is guaranteed to stay in the client cache until timestep C, it will provide a constant amount of utility gain through time interval $[t, C]$. Note that we dropped $s_t$ from P since, from the perspective of the scheduler, it is a fixed prediction.

$$U_{i,j}^t = \sum_{k=t}^{C} g^{k-1} P(q_i|k) g(j)$$

We denote by $f_{i,j}^t$ a binary variable that indicates if the j-th block of $q_i$ is sent at time interval $[t-1, t]$. With this notation, we can transform the objective into a linear function:

$$\sum_{k=1}^{C} \left( g^{k-1} \sum_{i=1}^{n} U(B_i^k) P(q_i|k) \right) = \sum_{i=1}^{n} \sum_{j=1}^{N_b} \sum_{k=1}^{C} f_{i,j}^k U_{i,j}^k \quad (3)$$

**Constraints.** Our ILP program must account for three constraints. The ring buffer's limited capacity is implicitly encoded in our choice of maximum time horizon C in the objective. The ILP hard constraints ensure that (1) the network bandwidth is not exceeded, and that (2) each block is only sent once:

$$\forall k \sum_{i,j} f_{i,j}^k \leq 1 \qquad \forall i,j \sum_{k} f_{i,j}^k \leq 1$$

**Limitations.** The LP scheduler is very slow because the LP problem size, as well as the cost to compute the utility gain matrix $U_{i,j}^t$, increases with the time horizon (cache size), the interaction space (number of possible requests), and the granularity of the progressive encoding (number of blocks). For instance, if the image application (10k possible requests) has a cache size of 5k blocks, and 10 blocks per request, the LP will contain 0.5 billion variables. Simply constructing the problem in the appropriate format for a solver is too expensive for a real-time system, and further, this cost is incurred for every C blocks to be sent. The technical report [47] details micro-experiments, including comparisons with the fast, greedy scheduler described next.

## 5.3 Greedy Scheduler

This subsection describes KHAMELEON's fast greedy scheduler (Listing 1). The main design consideration is that it can rapidly make scheduling decisions as the client sends distributions at unpredictable rates, and without blocking (or being blocked by) the sender. We first describe the scheduler design, and then discuss the interaction between the scheduler and the sender. The technical report [47] describes the formal semantics of a correct schedule, given a sequence of distributions sent from the client.

### 5.3.1 Greedy Scheduler Design

Our greedy scheduler uses a single-step horizon (first term in Equation (1)). It computes the expected utility gain for giving one block to each request (accounting for the number of blocks that have already been scheduled), and samples a request $q_i$ proportional to its utility gain. The next block is allocated to $q_i$. It schedules batches of C blocks to fully fill the client cache, then it resets its state and repeats.

**State.** The algorithm keeps three primary pieces of state that vary over time. The first is the number of blocks assigned to each request $B = [b_1, \ldots, b_n]$. This is used to estimate the utility gain for the next scheduled block, and is reset after a full schedule (C blocks) have been scheduled. The second state materializes g() as an array. The third state precomputes the matrix $\mathbb{P}_{i,t} = \int_{k=t}^{C-1} P(q_i|k)$ that represents the probability that the user will request $q_i$ over the rest of the batch. This is estimated as a Reimann sum via the Trapezoidal Rule (lines 8-11).

Scheduling is now a small number of vectorized operations. The expected utility gain at timestep t is the dot product $\mathbb{P}_t \bullet g[B]$, where $\mathbb{P}_t = [\mathbb{P}_{1,t}, \ldots, \mathbb{P}_{n,t}]$ and $g[B] = [g(b_1), \ldots, g(b_n)]$ are vectorized lookups (line 16).

**Scheduler Algorithm.** The client is allowed to send new probability distributions at arbitrary times. If a new distribution arrives, we wish to use its more accurate estimates, but also do not wish to waste the resources used for prior scheduling work. Further, the scheduler should progress irrespective of the rate at which the client sends distributions.

To make progress, each iteration schedules up to a batch of `bs` blocks at a time (default of 100). After each batch, it checks whether a new distribution has arrived, and if so, recomputes the $\mathbb{P}_{i,t}$ matrix (lines 6-11). Since t blocks may already have been chosen for the current schedule, we only need to materialize the time slots for the rest of the schedule ($\mathbb{P}_{i,t'}$ where $t' \in [t+1, C-1]$). After sending the scheduled blocks to the sender, it resets t and B if a full schedule has been allocated (lines 21-23).

**Optimizations.** We employ several straightforward optimizations beyond the pseudocode in Listing 1. The main one avoids materializing the full $\mathbb{P}_{i,t}$ matrix when the number of requests is very high. Most requests will have the same probability of $\approx 0$ (images 4, 8, 9, 12, 13-16 in Figure 4), and correspondingly similar utility gains. Thus, we group these requests into a single "meta-request" whose probability is the sum of the individual requests. If the scheduler samples this meta-request in line 17, then it uniformly chooses one of the individual requests. On a benchmark with 10K requests, 5K blocks in the cache, and 50 blocks per request, this optimization reduces the cost of generating one schedule from 1.9s to 150ms (13× reduction). Using this concept to further stratify the probability distribution may further reduce runtime, but we find that this optimization is sufficient in our experiments.

```
1   C, g, n     // cache size, utility array, # requests
2   bs = min(C,bs)  // blocks to schedule per iter
3   B = [0,..,0]    // # blocks for each req in cache
4   t = 0           // # blocks scheduled
5   while True:
6     if received_new_distribution()
7       dist = get_new_distribution()
8       for i∈[1, n]
9         ℙ_i,C = dist(i, C)
10        for t'∈[C-1, t]
11            ℙ_i,t' = 1/(t'+1) ℙ_i,t'+1 + t/(t'+1) dist(i,t')
12
13    S = [ ]          // generated batch of blocks
14    while t < C-1 and |S| < bs
15      t += 1
16      u = ℙ_t ● g[B]
17      q = sample requests proportional to u
18      S.append(q)
19      B[q] += 1
20
21    send S to sender
22    if t == C      // reset after a full schedule
23      t,B = 0, [0,..,0]
```

Listing 1: Pseudocode of the greedy scheduler algorithm.

### 5.3.2  Sender Coordination

Our current prototype assumes that the client and server clocks are synchronized to ensure that servers can ensure sufficient confidence in predictions, and that the Sender thread can be preempted. When a new prediction arrives at the scheduler, it identifies the position $i$ of the sending thread in the current batch, and reruns the scheduler from $i$ to $C$. The blocks for $0$ to $i$ do not change since they have already been sent. This is analogous to setting $t = i$.

The scheduler sends this partial schedule to the sending thread, which in the meantime, may have sent an additional $h$ blocks. Thus, it simply starts sending using the partial schedule at $i + h$. Concurrently, the scheduler begins scheduling the next batch using the updated predictions. Note that the scheduler may be modified to match a different client-cache replacement strategy; we leave this to future work.

### 5.4  Implementation Details

**Bandwidth Estimation.** The sender thread and scheduler require knowledge of the available network bandwidth, and aim to run at a rate that will not cause congestion on the network. KHAMELEON is agnostic to the specific end-host bandwidth estimation (and prediction) technique that is used to compute this information [82, 81, 35]. Further, note that KHAMELEON can alternatively be configured to use a user-specified bandwidth cap (e.g., to comply with limited data plans). In our implementation, the KHAMELEON client library periodically sends its data receive rate to the server; the server uses the harmonic mean of the past 5 rates as its bandwidth estimate for the upcoming timestep, and aims to saturate the link. This approach capitalizes on recent observations that bandwidth can be accurately estimated over short time scales, particularly in backlogged settings that avoid transport layer inefficiencies (e.g., TCP slow-start-restart) [44] that mask the true available bandwidth at the application layer [82].

**Backend Scalability.** This work assumes that backend query execution is scalable, i.e., data stores can execute
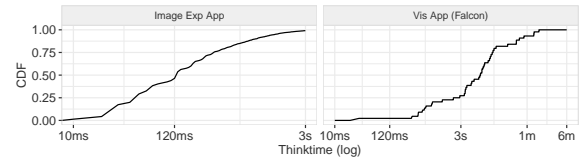


Figure 5: CDF of think times (time between consecutive requests) over interaction traces for the image and vis applications (§2).

many concurrent speculative requests without performance degradation. This is often true for key-value-oriented backends [78] or cloud services, but may not hold for other data stores. For instance, databases such as PostgreSQL have a concurrency limit, after which per-query performance suffers. Thus, it is crucial for the scheduler to avoid issuing too many speculative requests such that the backend becomes a bottleneck in response latency.

Although formally addressing this interaction challenge between KHAMELEON and data processing backends is beyond the scope of this paper, we use a simple heuristic to avoid triggering congestion in the backend. We benchmark the backend offline to measure the number of concurrent requests $C$ that it can process scalably. Let $n$ be the number of requests the backend is currently processing; we post-process schedules to ensure that they do not refer to blocks from more than $C - n$ distinct requests. In essence, we treat backend request limits in the same way as network constraints.

## 6.  EXPERIMENTS

We evaluate KHAMELEON on the DVE applications described in §2. Our experiments use real user interaction traces and a wide range of network and client cache conditions. We compare against idealized classic prefetching and response tuning approaches, highlight the benefits provided by each of KHAMELEON's features, and investigate KHAMELEON's sensitivity to environmental and system parameters. The results comparing KHAMELEON with the baselines are consistent across the applications. Thus, for space constraints, we primarily report results for the image application, and use Falcon to illustrate how KHAMELEON goes well beyond the state-of-the-art hand-optimized implementation (§6.4).

### 6.1  Experimental Setup

Our prototype uses a Typescript client library and a Rust server library. The client periodically sends predictions to the server every 150ms. Each prediction consists of a distribution over the possible requests at timesteps $D = \{50, 150, 250, 500ms\}$ from the time that the prediction is made; the 500ms values follow a uniform distribution. The application backends precompute and progressively encode results for all possible requests.

We use the image exploration and visualization applications described in §2. For the image application, we collected mouse-level interaction traces from 14 graduate students that freely used a version of the application that was configured with no response latency. Each trace is 3 minutes long, with 20ms average think time. For Falcon, we used the 70 traces from [6]. The interface used to collect these traces differs from the interface in the Falcon paper [49] by one chart (a bar chart instead of the heat map in [49]). Thus we translated the interactions over the bar chart to generate semantically

equivalent requests consistent with [49]. In this way, the performance numbers are comparable with [49]. We find that increasing the number and length of traces doesn't affect our results; Figure 5 reports the think-time distributions.

**Performance metrics:** KHAMELEON balances response latency and quality for preemptive interactions. However, due to the bursty nature of interactions, some requests (and their response data) may be preempted when later requests receive responses sooner. Thus, we report the percentage of preempted requests. For the non-preempted requests, we measure the cache hit rate as the requests that have blocks in the cache at the time of the request, the response latency as the time from when a request is registered with the cache to the upcall when one or more blocks are cached, and the response utility at that time. We will also evaluate how quickly the utility is expected to converge to 1 (all blocks).

We use the utility curves in Figure 3. The image application's utility function is described in §3.3. Falcon implements progressive encoding by sampling rows of the response in a round-robin fashion. For instance, for a 1D CDF, we sample values along the x-axis. We conservatively use the default linear utility function.

**Environment parameters:** Our experiments consider a wide range of network and client-side resource scenarios. We first use netem [51] to consider fixed bandwidth values between 1.5–15MB/s [4] (default 5.625MB/s) and request latencies between 20–400ms (default 100ms); note that because we precompute all responses, request latency is meant to include both network latency (between 5–100ms) and simulated backend processing costs (15–300ms). We vary the client's cache size between 10–100MB (default 50MB). We also use the Mahimahi network emulator [54] to emulate real cellular links; in these experiments, the minimum network round trip time was set to 100ms [55]. We simulate varying think time between requests from 10–200ms, which is favorable to the baseline approaches described below. Figure 5 shows CDFs of think times in our user traces.

**Performance baselines:** BASELINE is a standard request-response application with no prefetching. PROGRESSIVE mimics BASELINE, but only retrieves the first block of any response—this is intended to reduce network congestion but does not use prefetching to mask request latency.

Prefetching techniques primarily focus on prediction accuracy and the number of parallel requests to make. Modern predictors exhibit ≤ 70% accuracy [5]. To create strong baselines (`ACC-<acc>-<hor>`), we use a perfect predictor that knows the next `hor` requests with `acc` accuracy per request. After each user-initiated request, the prefetcher issues up to `hor` prefetching requests; to avoid triggering network congestion, it does not prefetch if the number of outstanding requests will exceed a bandwidth-determined threshold. For example, after the $i^{th}$ user request, `ACC-.8-2` will predict the $i+1^{th}$ and $i+2^{th}$ requests, and each will have 80% chance of being correct (i.e., matching the actual request in the trace). We use `ACC-0.8-1`, `ACC-1-1`, and `ACC-1-5` (following [5]). All baselines use an LRU cache.

We also evaluate an ORACLE version of KHAMELEON where the predictor knows the exact position of the mouse after D milliseconds (by examining the trace).
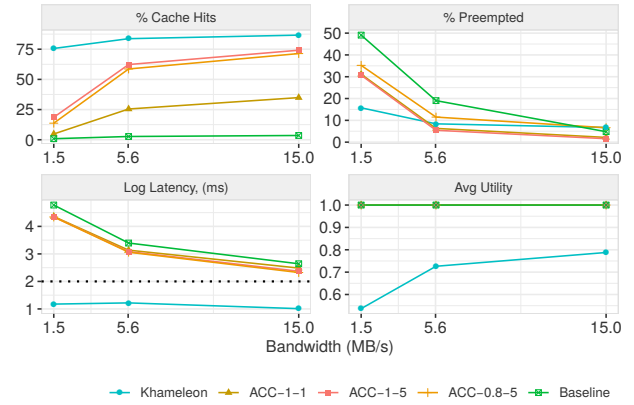


Figure 6: Idealized prefetching baselines and KHAMELEON for 50MB cache size and varying network bandwidths (x-axis). Pane headers list the metric for each y axis. Latency charts in all figures render a black dashed line at 100ms.

## 6.2 Comparison with Baselines

We first compare KHAMELEON with the aforementioned baselines including no prefetching, and `ACC-0.8-1`, `ACC-1-1`, and `ACC-1-5`. Recall that these are upper bounds for existing prefetching approaches—typical predictors have accuracies of < 70%.

**Varying Bandwidth and Cache Resources:** We faithfully replayed the user traces, and varied the cache size (10–100MB) and bandwidth resources (1.5–15MB/s), while keeping request latency fixed to 100 ms. For space reasons, Figure 6 shows results for medium cache size (50MB), since the findings are the same across cache sizes. The top row reports the percentage of requests for which one or more blocks are present in the cache at the time of request (i.e., % Cache Hits), and the percentage of preempted requests. KHAMELEON increases the cache hit rate by 24.5 − 88× above BASELINE, and by 1.17 − 15.82× above the idealized prefetching baselines. KHAMELEON reduces the number of preempted requests by 3× in low bandwidth settings, and has slightly higher preemption rate than `ACC-*-*` at higher bandwidths because its high cache hit rate causes more out-of-order responses. The `ACC-*-*` baselines have lower cache hits because think times are lower than request latency, thus the user has moved on by the time the prefetched data arrives.

The bottom row plots the utility and user-perceived response latency for requests that are *not preempted*. We see that the baselines consistently prioritize full responses—their utilities are always 1 at the expense of very long response latencies (note latencies are log scale). In contrast, KHAMELEON gracefully tunes the utility based on resource availability—all the while maintaining consistently low average response latencies that do not exceed 14ms across the bandwidth and cache size configurations. **On average, across different cache resources and bandwidth limits, Khameleon has up to 16× better cache hit rates than `ACC-*-*`, resulting in 16.35−1525.23× lower response times.**

To better illustrate the tradeoff between resources, responsiveness, and utility, Figure 7 compares average response latency (across all requests) and the response utility, for every condition (shape, color), bandwidth (size), and cache size; upper left means faster response times and higher utility.

---

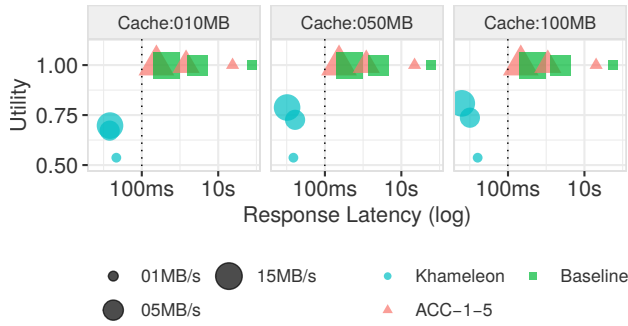[4]We report the bandwidth as `MB/s` instead of `Mb/s` to use the same units as block sizes.

Figure 7: Response time vs utility for the prefetching baselines and KHAMELEON. Size denotes bandwidth; upper left is better. Black dotted line shows 100ms latency threshold.
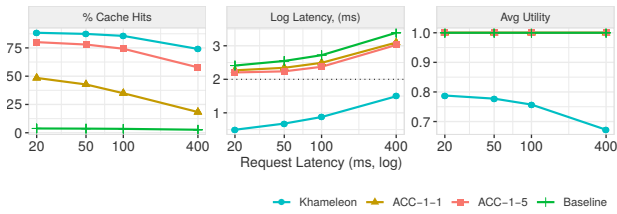


Figure 8: KHAMELEON vs prefetch baselines across varying request latencies; request latency includes both network and server processing delays.
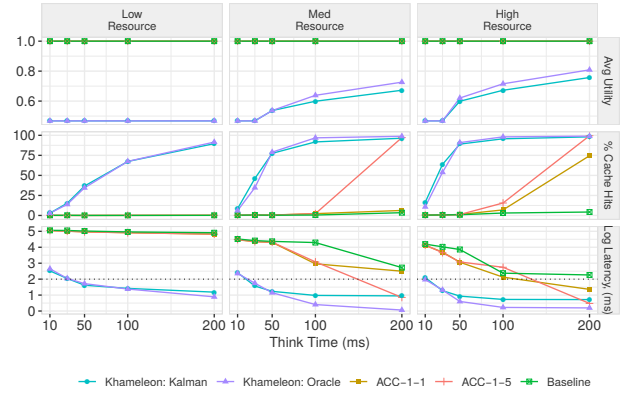


Figure 9: Varying think time between consecutive requests. Comparing KHAMELEON vs ACC using perfect and kalman filter predictors for 1 and 5 request horizons.
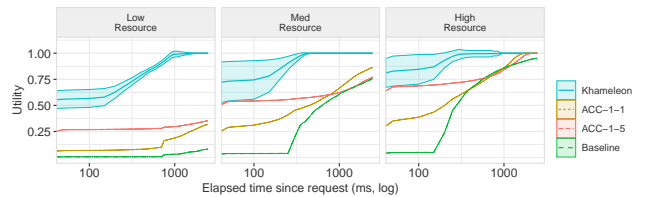


Figure 10: Convergence rate of utility: average utility (y-axis) over time after the user stops on a request.

Across all conditions, increasing the bandwidth improves the response times. However, the baselines remain at perfect utility and have high latencies. In contrast, KHAMELEON always has $< 100$ms latency and judiciously uses resources to improve utility.

**Request latency:** We now fix network bandwidth (15MB/s) and cache size (50MB), and vary request latency (20–400ms). Recall that request latency includes both network and server processing delays. Figure 8 shows that KHAMELEON consistently achieves higher cache hit rates than the prefetching baselines. As request latencies grow, KHAMELEON degrades response utility to ensure that response latencies remain low (on average 11ms). In contrast, the alternatives pursue perfect utilities at the detriment of responsiveness. When the request latency is 400ms, Khameleon performs $79\times$ faster than BASELINE, and $37\times$ than ACC-*-*. The baselines become highly congested as the request latency increases.

**Think time:** So far, we have faithfully replayed the user traces. Now we synthetically vary the think times in the traces between 10–200ms to assess its effect. We fix request latency to 100ms, and use three resource settings: low (bandwidth=1.5MB/s, cache=10MB), medium (5.625MB/s, 50MB), and high (15MB/s, 100MB).

Figure 9 indeed shows that high think times improve all prefetching methods by reducing congestion and giving more time to prefetch. This is most notable in the high resource setting, where the BASELINE response latency (bottom row) converges to the cost of the network latency plus the network transmission time. ACC-1-* has high response latency when the think time is short due to congestion, but the cache rate increases to $75-100\%$ with high think time and high resources. With low resources and low think times, KHAMELEON achieves low latency by hedging, as shown by

the low utility values. Despite this, the next experiment shows that KHAMELEON converges to full utility faster than the baselines. With more resources, KHAMELEON shifts to prioritize and improve utility. We find that KHAMELEON is close to ORACLE, except in high resource settings, where perfect prediction can better use the extra resources and further reduces latency by $2\times$. **Khameleon maintains near-instant response latencies, and uses the additional think time to increase the response utility.** This highlights KHAMELEON's efficacy for DVE applications with low think times relative to request latency, i.e., where there is not enough time to prefetch between requests, even with perfect prediction.

**Convergence:** Although trading utility for responsiveness is important, the response should quickly converge to the full utility when the user pauses on a request. We now pause a user trace at a random time, and track the upcalls until the utility reaches 1. We use the high, medium, and low resource settings described above. Figure 10 reports the average and standard deviation of the utility after waiting an elapsed time after pause.[5] KHAMELEON consistently converges to a utility of 1 faster than all of the baselines, in expectation. This is explained by the additional congestion incurred due to the high rate of requests issued by the two baselines. We expect that better application-specific predictors [5] can greatly improve convergence for KHAMELEON.

## 6.3 Understanding Khameleon

We now perform an ablation study, and vary system configurations to better understand KHAMELEON.

**Ablation Study.** To show the importance of jointly optimizing resource-aware prefetching and progressive response

---
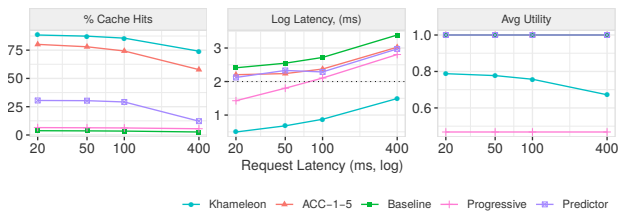[5]Utility baselines are 0 or 1, so we report the average.
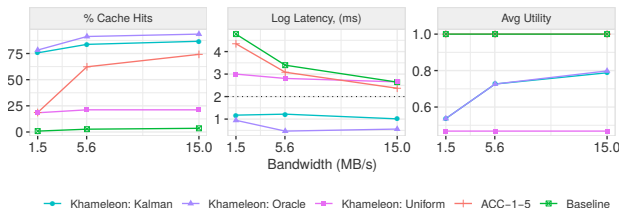
Figure 11: Results of ablation study.



Figure 12: Varying KHAMELEON predictors.



Figure 13: Performance on time-varying cellular networks.

encoding, we perform an ablation study. Starting with a non-prefetching BASELINE, we add the kalman filter and joint scheduler but without progressive encoding (PREDICTOR), and we add progressive encoding but without prefetching to show the benefits of cache amplification (PROGRESSIVE). For reference, we compare with `ACC-1-5`. We use a bandwidth of 15MB/s, cache size of 50MB, and vary request latencies.

In Figure 11, increased request latency lowers the cache hit rate for all approaches (negligible for KHAMELEON). PREDICTOR improves over BASELINE because the joint scheduler pushes predicted requests proactively (thus increasing the cache hit rate) without increasing network congestion. PROGRESSIVE improves over BASELINE by reducing the network transmission time and alleviating congestion, yet its utility is also the lowest. The combination of the two optimizations are needed in KHAMELEON for higher utility, consistently $< 31$ms response, and ($> 74\%$) cache hit rate.

**Sensitivity to Predictors.** Figure 12 assesses the impact of the predictor by comparing the UNIFORM predictor, KALMAN, and the ORACLE predictor as the upper bound. We fix request latency to 100ms, and include `ACC-1-5` and BASELINE as references. At low bandwidth, simply using the KHAMELEON framework already improves latency compared to `ACC-1-5`; and KALMAN further improves on top of UNIFORM and is close to ORACLE. As bandwidth increases, a more accurate predictor better uses the resources to push more blocks for more likely requests. Thus, ORACLE further reduces response times by $1.7 - 5.7\times$ compared to KALMAN.

**System Parameters and Bandwidth Overheads.** We find that the client can send prediction distributions every 50–350m without affecting KHAMELEON performance, but lower frequencies degrade performance. KHAMELEON overpushes (% of blocks sent but unused) $50 - 75\%$ of the blocks, as compared to $35 - 45\%$ for `ACC-1-5`. We believe these rates are acceptable given the orders of magnitude lower latency and because the user can limit the bandwidth allocated to prefetching. Details are in the technical report [47].

**Real Network Traces.** Using Verizon and AT&T LTE cellular network traces, with a fixed 100ms request latency and 50M cache size, Figure 13 shows that KHAMELEON considerably outperforms `ACC-1-5`. The cache hit rate is over $10\times$ higher on AT&T, and the latency is lower by $348.36-430.12\times$.
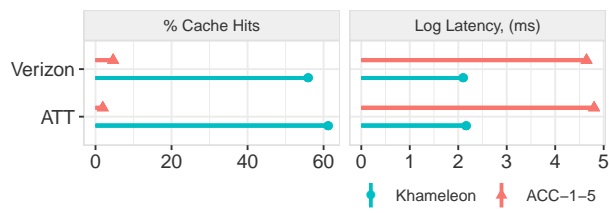
## 6.4 Falcon Visualization Experiments

We now adopt Falcon [49] to KHAMELEON, and show that the ability to easily change the predictor and introduce progressive encoding lets KHAMELEON further improve over the already–optimized Falcon.

**Porting Falcon:** We modified the Typescript client to register requests to the KHAMELEON client library. Originally, when the user hovers over a chart, Falcon issues 5 separate SQL queries to the backend database to generate a data slice for each of the other five charts (we call this group of queries a single *request*). We simulate this with a predictor that assigns a probability of 1 to the currently hovered upon view, and 0 to all others. Similarly, when the scheduler allocates one block for a given request, the sender issues 5 queries to the query backend (PostgreSQL database), and progressively encodes the combined query results into blocks. In contrast to the image application, the backend only scales to 15 concurrent queries before query performance degrades considerably. Thus, prefetching even 3 requests can issue enough queries to saturate the backend.

Adapting the client required $\approx 50$ LOC—mostly decoding blocks into Falcon's expected format. The code to encode query results into blocks on the server required $\approx 60$ LOC.

**Experiment Setup:** We create two databases using subsets of the flights dataset from Falcon [49]; SMALL has 1M records with query latencies of $\approx 800$ms, and BIG has 7M records with latencies of $\approx 2$s. We verified that the ported version performed the same as the original Falcon, so we report metrics based on varying the ported version. We report results on BIG, since SMALL exhibits the same trends. The full results are presented in the technical report [47].

**Predictor and Progressive Encoding:** We change the predictor from Falcon's "on-hover" (dashed lines) to the kalman filter (solid lines) used in earlier experiments. The x-axis varies the number of blocks that each request is encoded into (each block has fewer result records). The red lines in Figure 14 (PostgresSQL) show that KALMAN improves over ONHOVER, delivering $1.4\times$ more cache hits, $4\times$ lower latency on average, and higher utility, particularly as the number of blocks increases.

**Scalable Backend:** We now simulate a scalable database (blue lines). We first precompute and log each query's execution times when running in isolation. The backend answers queries from a cache and simulates the latency. Compared to the PostgreSQL backend, KALMAN response latencies improves by $2.6\times$, and ONHOVER by $1.3\times$. KALMAN still outperforms ONHOVER with higher utility because it hedges more aggressively without congesting the backend.
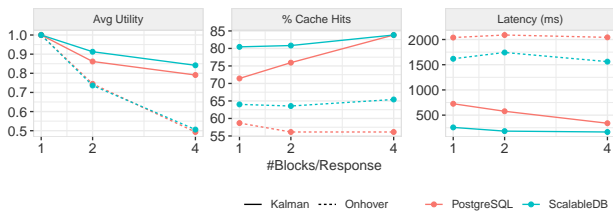
Figure 14: Ported Falcon system varying number of blocks/request (x-axis), and predictors (line) using Postgres (red line) and a simulated scalable database backend (blue line).

# 7. RELATED WORK

**DVE application optimizations.** Many approaches reduce response latencies by addressing server-side data processing and client-side rendering costs. Modern Databases can run queries in tens of milliseconds using precomputation (of datacubes [28], samples [19], indexes [22]), hardware [45, 27] or vectorized [83] acceleration, and parallelization [78]. Client-side computation and rendering delays can be reduced via datacubes [40] and GPU acceleration [27, 42, 46].

KHAMELEON is complementary to the above approaches and focuses on reducing the network bottlenecks (not client- or server-side computation delays) in DVE applications. Indeed, KHAMELEON could be used as the communication layer to progressively encode and push optimized data structures and query results based on anticipated user interactions. Our current implementation makes the simplifying assumption that data processing and progressive encoding incur a fixed cost; leveraging the above query processing and response encoding optimizations will require incorporating data processing costs into the scheduler, a promising future direction.

**Caching and Prefetching.** Interactive data exploration has studied caching architectures [69] and prefetching techniques to pre-populate caches [11, 21, 2, 5]. ATLAS [11] extrapolates scrollbar movements to prefetch subsequent requests; ForeCache [5] extends this to prefetch tiles of array data by forecasting future queries based on past user actions and statistical data features; These approaches are crafted for their specific visualization interface, and leverage restrictions of the user's allowed interactions (often to linear slider actions or panning to nearby tiles) that help improve the prediction accuracy. Falcon [49] prefetches when a user hovers over a chart, and uses the time between hovering and interacting with the chart to prefetch datacube structures. Database query prefetching typically relies on Markov models that update when a new query is issued [62, 66, 36, 10], which assumes longer think times between queries, while web page requests [20, 50, 75] use mouse predictors similar to the Kalman Filter used in the experiments.

Though these techniques are able to perform backend query computation early, they do not incorporate server push mechanisms or progressive response encoding, limiting their impact on alleviating network bottlenecks. KHAMELEON borrows similar prediction information, but replaces explicit requests and responses with probability distributions and a fine-grained scheduler for push-based streaming that accounts for request probabilities *and* response quality.

**Progressive Encoding.** Progressive encoding ensures that a small prefix of a response is sufficient to render an approximate result, and additional blocks improves result quality (ultimately converging to the fully accurate version) [33]. This is applied to a wide range of data, including images [65, 68], layered encodings [34, 67, 26, 15], visualization data [5], and webpage resources [53, 52, 64]. KHAMELEON lets applications provide progressively encoded responses [65, 68, 5] so the scheduler joint optimization can dynamically balance response quality for low latency in DVE applications.

**Progressive Computation.** Online aggregation [32, 1, 39, 3, 61] and progressive visualization [23, 48, 24, 70, 58] quickly return approximate results whose estimates improve over time, and could be backends in KHAMELEON. DICE [37] also uses speculation to accelerate visualization exploration, but is limited to faceted data cube navigation. It speculatively executes approximate queries for neighboring facets using a sharded database, and allocates sampling rates to the queries based on the expected accuracy gains.

Progressive computation is fundamentally different than the *progressive encoding* used by KHAMELEON. In the former, each improvement is *a separate result set*—it sends more data and uses more network resources. Each result set can be progressively encoded and benefit from KHAMELEON. Thus, although their *effects* both progressively enhancing rendered data, the mechanisms are different and complementary.

# 8. DISCUSSION AND CONCLUSION

KHAMELEON is a dynamic prefetching framework for data visualization and exploration (DVE) applications that are approximation tolerant. Rather than focusing solely on predicting requests to prefetch or adapting response quality to available resources, KHAMELEON uses a server-side scheduler to jointly optimize across these techniques. Responses are progressively encoded into blocks, and proactively streamed to the client cache based on request likelihoods.

KHAMELEON consistently achieves sub-30ms response times even when requests take 400ms, and out-performs existing prefetching techniques (often by OOM). It gracefully uses resources to improve quality. To best leverage KHAMELEON, each component in the system (the backend scalability, network bandwidth, degree of speculative prefetching) should be matched to produce and consume data at the same rates.

**Learning Improved Policies.** This work used naive predictors and scheduling policies to argue for the effectiveness of a continuous push framework. As expected, we also found a considerable gap from an optimal predictor; we expect better scheduling policies as well. One extension is to adapt a Reinforcement Learning framework to improve the scheduler's policy. For instance, we could log explicit reward signals in the client, and use Q-learning to better estimate future rewards. We could also unroll beyond a single step, and use policy gradient methods [71] to learn a higher quality policy function that may account for deployment idiosyncrasies. The challenge is to balance more sophistication with the need to schedule the next block in real-time (microseconds).

# 9.  REFERENCES

[1] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica. Blink and its done: Interactive queries on very large data. *PVLDB*, 5(12):1902–1905, 2012.

[2] Z. Ahmed and C. Weaver. An adaptive parameter space-filling algorithm for highly interactive cluster exploration. In *VAST*, 2012.

[3] D. Alabi and E. Wu. Pfunk-h: Approximate query processing using perceptual models. In *HILDA*, 2016.

[4] G. A. Aydemir, P. M. Langdon, and S. Godsill. User target intention recognition from cursor position using kalman filter. In *Conf. on Universal Access in Human-Computer Interaction*, 2013.

[5] L. Battle, R. Chang, and M. Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *SIGMOD*, 2016.

[6] L. Battle, P. Eichmann, M. Angelini, T. Catarci, G. Santucci, Y. Zheng, C. Binnig, J.-D. Fekete, and D. Moritz. Database benchmarking for supporting real-time interactive querying of large data. In *SIGMOD*, 2020.

[7] L. Battle and J. Heer. Characterizing exploratory visual analysis: A literature review and evaluation of analytic provenance in tableau. In *Comput. Graph. Forum*, 2019.

[8] R. Begleiter, R. El-Yaniv, and G. Yona. On prediction using variable order markov models. In *JAIR*, 2004.

[9] Human cell atlas. `https://chanzuckerberg.com/science/programs-resources/humancellatlas/`, 2018.

[10] U. Cetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query steering for interactive data exploration. In *CIDR*, 2013.

[11] S.-M. Chan, L. Xiao, J. Gerth, and P. Hanrahan. Maintaining interactivity while exploring massive time series. In *VAST*, 2008.

[12] Y. Chen, A. Rau-Chaplin, et al. cgmOLAP: Efficient Parallel Generation and Querying of Terabyte Size ROLAP Data Cubes. In *ICDE*, 2006.

[13] Art institute of chicago: The collection. `https://www.artic.edu/collection`, 2018.

[14] C. Cruz-Neira, D. Sandin, and T. A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the cave. In *SIGGRAPH*, 1993.

[15] D. Dardari, M. G. Martini, M. Mazzotti, and M. Chiani. Layered video transmission on adaptive ofdm wireless systems. In *EURASIP*, 2004.

[16] J. Deber, R. Jota, C. Forlines, and D. J. Wigdor. How much faster is fast enough?: User perception of latency & latency improvements in direct and indirect touch. In *CHI*, 2015.

[17] J. DeBrabant, C. Wu, U. Cetintemel, and S. Zdonik. Seer: Profile-driven prefetching and caching for interactive visualization of large multidimensional data. Technical report, 2015.

[18] E. Dimara and C. Perin. What is interaction for data visualization? In *TVCG*, 2020.

[19] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample+ seek: Approximating aggregates with distribution precision guarantee. In *SIGMOD*, 2016.

[20] J. Domènech, J. A. Gil, J. Sahuquillo, and A. Pont. Web prefetching performance metrics: A survey. In *Performance Evaluation*, 2006.

[21] P. R. Doshi, E. A. Rundensteiner, and M. O. Ward. Prefetching for visual data exploration. In *DASFAA*, 2003.

[22] M. El-Hindi, Z. Zhao, C. Binnig, and T. Kraska. Vistrees: fast indexes for interactive data exploration. In *HILDA*, 2016.

[23] J.-D. Fekete. Progressivis: A toolkit for steerable progressive analytics and visualization. In *DSIA*, 2015.

[24] D. Fisher, I. Popov, S. Drucker, et al. Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster. In *CHI*, 2012.

[25] Geotiff. `http://trac.osgeo.org/geotiff/`, 2018.

[26] M. M. Ghandi, B. Barmada, E. V. Jones, and M. Ghanbari. H.264 layered coded video over wireless networks: Channel coding and modulation constraints. In *EURASIP*, 2006.

[27] Graphistry: Find the stories in your data. `https://www.graphistry.com`, 2018.

[28] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Data Mining and Knowledge Discovery*, 1995.

[29] Q. Guo and E. Agichtein. Exploring mouse movements for inferring query intent. In *SIGIR*, 2008.

[30] M. Haklay and P. Weber. Openstreetmap: User-generated street maps. In *IEEE Pervasive Computing*, 2008.

[31] Q. He, D. Jiang, Z. Liao, S. C. Hoi, K. Chang, E.-P. Lim, and H. Li. Web query recommendation via sequential query prediction. In *ICDE*, 2009.

[32] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD Record*, 1997.

[33] H. Hoppe. Progressive meshes. In *SIGGRAPH*, 1996.

[34] S. Jakubczak and D. Katabi. A cross-layer design for scalable mobile video. In *MobiCom*, 2011.

[35] J. Jiang, S. Sun, V. Sekar, and H. Zhang. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *NSDI*. USENIX Association, 2017.

[36] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and Interactive Cube Exploration. In *ICDE*, 2014.

[37] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and interactive cube exploration. In *ICDE*, 2014.

[38] Nasa: Landsat science. `https://landsat.gsfc.nasa.gov`, 2018.

[39] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *SIGMOD*, 2016.

[40] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. In *TVCG*, 2013.

[41] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. In *Vis*. IEEE, 2014.

[42] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. In *Computer Graphics Forum*, 2013.

[43] H. S. Malvar. Fast progressive wavelet coding. In *DCC*, 1999.

[44] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *SIGCOMM*, 2017.

[45] Omnisci immerse. `https://www.omnisci.com/demos/`, 2018.

[46] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodık. Superconductor: A language for big data visualization. In *Workshop on Leveraging Abstractions and Semantics in High-Performance Computing*, 2013.

[47] H. Mohammed, Z. Wei, E. Wu, and R. Netravali. Continuous prefetch for interactive data applications. In *ArXiv*, 2020.

[48] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *CHI*, 2017.

[49] D. Moritz, B. Howe, and J. Heer. Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *CHI*, 2019.

[50] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. In *TKDE*, 2003.

[51] Netem - network emulator. `https://man7.org/linux/man-pages/man8/tc-netem.8.html`, 2011.

[52] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, 2016.

[53] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring time-to-interactivity for modern web pages. In *NSDI*, 2018.

[54] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for http. In *ATC*, 2015.

[55] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. Watchtower: Fast, secure mobile page loads using remote dependency resolution. In *MobiSys*, 2019.

[56] P. T. Pasqual and J. O. Wobbrock. Mouse pointing endpoint prediction using kinematic template matching. In *CHI*, 2014.

[57] M. Pintus, G. Ginesu, L. Atzori, and D. D. Giusto. Objective evaluation of webp image compression efficiency. In *MobiMedia*, 2011.

[58] M. Procopio, C. Scheidegger, E. Wu, and R. Chang. Load-n-go: Fast approximate join visualizations that improve over time. In *DSIA*, 2017.

[59] M. L. Puterman. Markov decision processes: Discrete stochastic dynamic programming. In *Wiley Series in Probability and Statistics*, 1994.

[60] P. Rahman, L. Jiang, and A. Nandi. Evaluating interactive data systems. In *VLDBJ*, 2019.

[61] S. Rahman, M. Aliakbarpour, H. K. Kong, E. Blais, K. Karahalios, A. Parameswaran, and R. Rubinfield. I've seen enough: incrementally improving

visualizations to support rapid decision making. *PVLDB*, 10(11):1262–1273, 2017.

[62] K. Ramachandran, B. Shah, and V. V. Raghavan. Dynamic Pre-Fetching of Views Based on User-Access Patterns in an OLAP System. In *SIGMOD*, 2005.

[63] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Give in to procrastination and stop prefetching. In *HotNets*. ACM, 2013.

[64] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *SIGCOMM*, 2017.

[65] D. Salomon and G. Motta. *Handbook of data compression*. Springer Science & Business Media, 2010.

[66] C. Sapia. PROMISE: Predicting Query Behavior to Enable Predictive Caching Strategies for OLAP Systems. In *DaWaK*, 2000.

[67] C. A. Segall and G. J. Sullivan. Spatial scalability within the h.264/avc scalable video coding extension. In *TCSVT*, 2007.

[68] J. M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. In *IEEE Transactions on signal processing*. IEEE, 1993.

[69] R. Sisneros, C. Jones, J. Huang, J. Gao, B.-H. Park, and N. F. Samatova. A multi-level cache model for run-time optimization of remote visualization. In *TVCG*, 2007.

[70] C. D. Stolper, A. Perer, and D. Gotz. Progressive visual analytics: User-driven visual exploration of in-progress analytics. In *TVCG*. IEEE, 2014.

[71] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, 1999.

[72] G. K. Wallace. The jpeg still picture compression standard. In *IEEE transactions on consumer electronics*. IEEE, 1992.

[73] J. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. In *IEEE Transactions on Image Processing*, 2004.

[74] G. Welch, G. Bishop, et al. An introduction to the kalman filter. 1995.

[75] R. W. White, F. Diaz, and Q. Guo. Search result prefetching on desktop and mobile. In *TOIS*, 2017.

[76] J. O. Wobbrock, J. Fogarty, S.-Y. S. Liu, S. Kimuro, and S. Harada. The angle mouse: target-agnostic dynamic gain adjustment based on angular deviation. In *CHI*, 2009.

[77] J. O. Wobbrock, A. D. Wilson, and Y. Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *UIST*, 2007.

[78] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A kvs for any scale. In *TKDE*. IEEE, 2019.

[79] Y. Wu, J. M. Hellerstein, and E. Wu. A devil-ish approach to inconsistency in interactive visualizations. In *HILDA*, 2016.

[80] Y. Wu, L. Xu, R. Chang, J. M. Hellerstein, and E. Wu. Making sense of asynchrony in interactive data visualizations. In *CoRR*, 2018.

[81] Q. Xu, S. Mehrotra, Z. Mao, and J. Li. Proteus: Network performance forecast for real-time, interactive mobile applications. In *MobiSys*. ACM, 2013.

[82] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Computer Communication Review*, 2015.

[83] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. In *IEEE Data Eng. Bull.*, 2012.