Getty Common Image Service

Research & Design Report

Stefano Cossu, J. Paul Getty Trust <scossu@getty.edu>

January-June 2019

Contents

1 Introduction **4**
1.1 Scope and Purpose 4
1.2 Project Description 4
1.3 Preexisting Status 4
1.4 Key Requirements 4

2 Architectural Components **5**
2.1 Image Formats and Encoding 5
2.1.1 Pyramidal TIFF 6
2.1.2 JPEG 2000 6
2.2 Image Processors 7
2.2.1 JP2 Processors 7
2.2.2 Pyramidal TIFF Processors 8
2.3 Image Server 9
2.3.1 Selected for Benchmark 10
2.3.2 Reviewed But Discarded 12
2.3.3 Not Reviewed 15
2.3.4 Information Sources 15
2.4 Gateway Service 17
2.4.1 Image API versioning 17
2.4.2 Caching 18
2.4.3 Derivative Cache 19
2.4.4 Web Front End & Load Balancing 20
2.5 Manifest Service 21
2.6 ETL And Migration 21
2.6.1 ETL Framework 22
2.6.2 Migration Scripts 22
2.7 Storage Technology and Vendors 22

3 Benchmarks **22**
3.1 Image Formats & Encoding 23
3.1.1 Test Sources and Setup 23
3.1.2 Test Results 23

3.2 Image Servers . . . 27
3.2.1 Reference Data Set . . . 27
3.2.2 Load Test Servers . . . 28
3.2.3 Benchmark Parameters . . . 28
3.2.4 Test Instruments and Methodology . . . 30
3.2.5 Test Result Summary . . . 30

4 Conclusions 40

5 Appendix 1: Recipes 42
5.1 JP2 encoding parameters . . . 42
5.2 Source Image Generation . . . 42

6 Appendix 2: Locust Test Data 43
6.1 IIPImage With PTIFF . . . 43
6.1.1 10 Connections . . . 43
6.1.2 100 Connections . . . 44
6.1.3 1000 Connections . . . 44
6.2 IIPImage With JP2 . . . 45
6.2.1 10 Connections . . . 45
6.2.2 100 Connections . . . 46
6.2.3 1000 Connections . . . 46
6.3 Cantaloupe . . . 47
6.3.1 10 Connections . . . 47
6.3.2 100 Connections . . . 48
6.3.3 1000 Connections . . . 48
6.4 Loris . . . 49
6.4.1 10 Connections . . . 49
6.4.2 100 Connections . . . 50

1 Introduction

1.1 Scope and Purpose

This document summarizes the research and design process supporting the implementation of the Getty Common Image Service (GCIS) as a whole and its individual components, and contains implementation guidelines for the project derived from such research.

Given the very large volumes of data that GCIS is meant to process and its key role in the Getty information architecture, a thorough study of the possible technical solutions for its various components was deemed necessary; hence the need for this document.

The information contained here is focused on the research made by the Getty for its own specific goals, IT infrastructure and resources, and is not intended to be a general solution to IIIF architectures. However, it is the author’s and the Getty’s hope that several other institutions may benefit from the publication of this information.

1.2 Project Description

The Getty Common Image Service (GCIS) is a suite of server software and data manipulation and management tools designed by Getty Digital (GDI) with the scope of providing a central facility for serving the Getty’s images on the Web.

GDI plans to take charge of serving millions of images from the various Getty programs using the IIIF protocol, by consolidating existing image services. The choice of an image server, data transformation pipelines and server infrastructure is critical to a successful implementation of this plan.

1.3 Preexisting Status

The Getty is currently adopting two separate IIIF services, maintained by two separate departments: one for the Museum collections, which have a Level 0 (static images + web server) setup, and one based on Loris for the Getty Research Institute (GRI).

Performance of the current setup is a concern, as well as the maintainability of the Level 0 implementation. The consolidation of the two systems into a larger one, maintained by GDI, should resolve both concerns.

1.4 Key Requirements

The planned system must fulfill the following key requirements:

1. **Stability**: The system as a whole must be able to support 24/7 operation under heavy load and spikes without downtime, and must have a very low (i.e. close to zero) request failure rate under normal operation conditions.
2. **Sustainability**: 3rd party software and code libraries used for in-house developed components should be open source, have a stable contributor base and be in a "stable" phase at least; i.e. their first "production-ready" release should have been released not too recently. This factor may be corroborated by the existence of notable, large-scale production implementations.
3. **Performance and Scale**: The server must be capable of handling very large volumes of traffic efficiently and continuously, use a reliable caching mechanism to reduce CPU cycles and network I/O when delivering standard derivatives, and respond well to load balancing. It must also allow software upgrades on individual systems without service downtime (i.e. all key systems must be redundant).
4. **Flexible storage**: IIIF source images will be stored in an AWS S3 bucket or a compatible store using the S3 API. The system must able to access these source images, possibly without any modification to the image server source code. Ideally, the architecture should be able to handle changes to the backend store protocol without major code changes.
5. **Stable IIIF Image API support**: Support for IIIF Image 2.x API, level 2 [1] and Presentation v2.1. In a not too far future, Image and Presentation 3.x API, as well as the other major API specs (Auth, Search), should be supported. It is expected that the software continue supporting newer versions of the API as these are published.
6. **Maintain embedded metadata**: One of the requirements for the images delivered by GDI is that they have some embedded EXIF metadata. The system must ensure that relevant metadata embedded in the original images are maintained in the final derivatives.

2 Architectural Components

This chapter will present the major decision areas for the GCIS architecture design. Tests and their results for these areas are described in the next chapter.

2.1 Image Formats and Encoding

Most IIIF image servers need images formatted and encoded with specific parameters in order to work efficiently. The choice of format and encoding scheme are key factors for performance and storage

[1] https://iiif.io/api/image/2.1/compliance/

efficiency.

Two image formats are viable with the majority of the image servers considered: pyramidal TIFF and JPEG2000.

The encoding scheme defines how image data are stored in the image file. Data can be uncompressed, or encoded with a lossless or a lossy compression scheme. Both TIFF and JPEG2000 support a number of encodings; given the large amount of data that we foresee to store, a scheme that is both space-efficient and CPU-efficient in the decoding phase is ideal. Encoding efficiency is not as critical as this is an operation that is done only once per image, while decoding happens much more frequently and is demand-driven; however, it plays a major role in large data migrations.

2.1.1 Pyramidal TIFF

Pyramidal TIFF (PTIFF) is part of the TIFF specification. It allows to embed multiple layers within an image, each with a different resolution. This has the advantage of having pre-compiled low resolution versions of the same image, so that if a thumbnail of the image is needed, only a smaller version of the image needs to be loaded. Internal layers can be compressed with a variety of lossless or lossy algorithms.

Pyramidal TIFFs need to be ordered in tiles, rather than in stripes in order to work with IIIF servers.

Note that, of all the servers tested, only IIPImage supports PTIFF at a competitive level.

PTIFF supports a number of encoding schemes, including very efficient ones such as JPEG and WebP. Many software libraries and applications (including IIPImage) use LibTIFF underneath, therefore can read these formats[2]. This makes PTIFF a valid alternative to JP2.

2.1.2 JPEG 2000

JPEG 2000 (JP2) is a newer standard than TIFF and extends the JPEG specifications. It is very space-efficient and reportedly optimized for partial image retrieval, which makes it particularly amenable for IIIF uses.

JP2 images can be compressed with the JPEG2000 scheme or uncompressed. JPEG2000 compression has been used in our benchmarks.

[2]Experiments with WebP-encoded PTIFF images and IIPImage have been conducted as part of this research and have proven that IIPImage is able to handle such images. However, these experiments were not thorough and did not measure decoding/encoding speed, image quality or file size scientifically.

2.2 Image Processors

There are open source and proprietary tools available to encode and decode JP2 and PTIFF images efficiently. These are used in the ETL phase that consists of converting source files into formats viable for the image server. Some of them are also used as decoders by some image servers.

The conversion from original sources to access masters includes several mandatory steps:

- Validate the image format, color model, depth, etc.
- Convert the source color profile to sRGB.
- Copy selected metadata from the source to the destination.
- Create progressively reduced resolutions as layers (pyramid), encoding each layer with the desired algorithm; or, in the case of JP2, create the corresponding image structure necessary for multi-resolution images.
- Save the file.

All of this needs to work within a Python framework.

2.2.1 JP2 Processors

The following tools were considered:

Kakadu

http://kakadusoftware.com/

Kakadu is the fastest JPEG2000 decoder today. A decoder is needed by most, if not all, image servers that need to generate derivatives from a JPEG2000 source.

Some image servers use Kakadu (and require a Kakadu license to run), some others use OpenJPEG and others yet let the implementer choose from either one.

Kakadu is commercial software and requires a license to be used on a public site. Evaluation licenses and Public Service licenses are available.

OpenJPEG

https://www.openjpeg.org/

OpenJPEG is a free and open source alternative to Kakadu to encode JPEG2000 images. Until recently, it was many times slower than Kakadu. However, recently a hefty effort was made to improve performance significantly. While it is reportedly still slower than Kakadu, OpenJPEG may be an alternative in situations where decoding performance may not be the only decision factor.

After some consideration, Kakadu was chosen without running any benchmark over OpenJPEG because the Getty already owns a Kakadu license and because we were quite confident about Kakadu's superior speed from other implementers' reports.

2.2.2 Pyramidal TIFF Processors

While the choice for processing JP2 images is quite straightforward, i.e. using Kakadu, generating PTIFFs is more complex. In spite of being an established standard, finding a toolkit to perform all the operations to create a pyramid file in an efficient and reliable way has proven to be one of the biggest challenges of this project. Several options were available, but all were missing something or were not reliable, or required an extensive setup; however, the additional effort to find an optimal production pipeline was justified by the long-term gain in processing efficiency over time.

The following tools were considered:

Libvips / PyVips

https://libvips.github.io/libvips/ Libvips is a demand-driven, horizontally threaded image processing library.

Vips has all the necessary functionality for the task, has Python bindings and runs very efficiently. However, over long-running processes it has been observed to consume an increasingly large amount of memory that does not get released, eventually leading to an application crash. The root cause for this is still being investigated.

LibTIFF, OpenCV, LittleCMS

LibTIFF is the most popular TIFF manipulation library. Recent versions of LibTIFF v4 include support for very efficient compression schemes such as JPEG and WebP.

The Python wrapper used here is tifffile which is a very minimal implementation which only support simple operations such as reading and writing files, encoding and decoding, etc. Operations such as resizing or color profile handling require additional tools.

This solution is a combination of several low-level libraries:

- libtiff to open the files, encode the derivative layers and save the pyramid layers.
- OpenCV to resize the image layers.
- LittleCMS to convert color profiles.

This is a much more laborious solution which requires handling images as raw arrays of numbers, interfacing with a C API (lcms) and building several libraries from source. It also runs as much as 10 times slower than Vips.

On the other hand, this option allows for very fine-grained control of the image quality, such as the resizing algorithm and compression schemes.

This method was devised as an alternative and comparison method to diagnose the above mentioned memory issue with Vips.

Pillow

Pillow is the principal image manipulation library for Python. It is reasonably efficient (much less than Vips though) and has a very extensive toolkit. The pyramid TIFF writing functionality is very obscure[3] but available.

Pillow offers all the functionality needed for the task, except for one: encoding the image in tiles, which is necessary for the access master format. This is a feature that is not yet implemented[4], which makes Pillow not viable for the task at the moment.

2.3 Image Server

The image server will not be developed in house and, ideally, should be a black box that GDI will only need to configure.

With this in mind, several IIIF Image API implementations have been reviewed and three have been selected for a performance conparison.

The following implementations, based on the Awesome IIIF list, have been reviewed:

- Cantaloupe
- Digilib
- Hymir

[3] https://github.com/python-pillow/Pillow/issues/2191#issuecomment-274285059
[4] https://github.com/python-pillow/Pillow/issues/672

- IIPImage
- Loris
- RAIS
- RIIF
- SIPI
- go-iiif
- iiif-s3

In-depth reports below are for a selected set of implementations (linked in the above list) that satisfy the key requirements previously indicated.

2.3.1 Selected for Benchmark

IIPImage

Facts

- IIIF Image API supported: 2.0
- Programming language: C, C++
- Project page: http://iipimage.sourceforge.net/
- Repo: https://github.com/iipimage/iipsrv
- Latest push to master or published release (whatever is most recent): 11/2017
- License: GPL v3
- Notable implementations: NGA, Bodleian (~1M), Wellcome Trust (36M)

Advantages

- Most mature of all server implementations
- Broadest adoption, even with the largest data sets known
- Very fast
- Support for special imagery, e.g. multi-spectral (may be an interesting feature for some scientific images from GRI or GCI)

Limitations

- Does not support S3 connection

- Only supports JP2 via Kakadu at the moment; OpenJPEG is in the plans.
- Limited resolver flexibility (only allows adding a prefix to map an identifier to a file on disk).

Cantaloupe

Facts

- IIIF Image API supported: 2.1
- Programming language: Java
- Project page: https://medusa-project.github.io/cantaloupe/
- Repo: https://github.com/medusa-project/cantaloupe
- Latest push to master or published release (whatever is most recent): 10/2018
- License: University of Illinois/NCSA Open Source License
- Notable implementations: YCBA

Advantages

- Very fast (but not as fast as IIPImage)
- Ruby API allows great flexibility in defining source image retrieval
- Excellent documentation

Limitations

- It's Java, with its memory management woes... However, in general this is a learning curve issue that stabilizes over time.

Loris

Facts

- IIIF Image APIs supported: 2.x (?)
- Programming language: Python
- Project page: https://github.com/loris-imageserver/loris
- Repo: https://github.com/loris-imageserver/loris
- Latest push to master or published release (whatever is most recent): 06/2018

- License: Other
- Notable implementations: GRI, Princeton Library

Advantages

- Written in Python, in theory allows for direct contribution to the software
- Reasonably fast[5]
- Can use Kakadu or OpenJpeg for decoding JP2

Limitations

- Previous experience showed high request failure rate in production
- Poor documentation
- Difficult setup
- No PTIFF support
- Python 2.x—limited in-house expertise

2.3.2 Reviewed But Discarded

SIPI

Facts

- IIIF Image API supported: Unkown
- Programming language: C++
- Project page: http://sipi.io
- Repo: https://github.com/dhlab-basel/Sipi
- Latest push to master or published release (whatever is most recent): 08/2018
- License: Affero GPL
- Notable implementations: DH Labs, University of Basel

Advantages

- Supposedly very fast (performance is one of the key goals for the project)

[5] Benchmarks between Cantaloupe and Loris were run at AIC and Loris turned out to be even faster under certain conditions.

- Configuration is in Lua and allows flexible mapping from URLs to source image locations and other complex setups
- Good documentation

Limitations

- Relatively new software; production-readiness, most importantly in high-load and high-volume scenarios, is unknown
- No known adopters other than its maintainer, the DH Labs of the University of Basel.
- Bundled with Kakadu, no other decoder (PTIFF, OpenJPEG) is possible.

Reason(s) For Discarding

- Narrow adopter base
- No significant advantages over other solutions

go-iiif

Facts

`go-iiif` is a fork of the Go `iiif` package.

- IIIF Image API supported: 2.1
- Programming language: Go (underlying C image library)
- Project page: https://aaronland.github.io/go-iiif/
- Repo: https://github.com/aaronland/go-iiif
- Latest push to master or published release (whatever is most recent): 07/2018
- License: BSD 3-Clause
- Notable implementations: unknown

Advantages

- S3 connector
- Uses libvips under the hood which is supposedly a very efficient image processing tool.

Limitations

- Only 2 significant contributors (apparently both from the same company)[6]. Moreover, the main maintainer left the project recently, which puts the project at serious risk of deprecation.
- Lots of add-ons (image filters, etc.) that we may not need and just add to the complexity of the code.
- The Go language itself is not yet as established as C/C++, Java or Python.

Reason(s) for Discarding

Lack of maturity and possible lack of developer support.

RAIS

Facts

- IIIF Image APIs supported: 2.1
- Programming language: Go
- Project page: https://github.com/uoregon-libraries/rais-image-server
- Repo: https://github.com/uoregon-libraries/rais-image-server
- Latest push to master or published release (whatever is most recent): 10/2018
- License: CC0 1.0
- Notable implementations: Historic Oregon Newspapers, UOregon Libraries

Advantages

- S3 connector
- Completely open source (uses OpenJPEG)

Limitations

- Cannot use Kakadu
- Only one maintainer
- Only one production use case

[6] https://github.com/aaronland/go-iiif/graphs/contributors

Reason(s) for Discarding

RAIS seems to have been conceived as a small-budget project with a specific use case in mind (the University of Oregon newspapers repository). Beyond that, and the availability of an S3 connector, RAIS does not seem to offer any outstanding feature compared to others.

RAIS is only maintained by one person, and has no other major adopter from what I could gather.

2.3.3 Not Reviewed

The following projects were not reviewed in depth, for the reasons explained below:

- Hymir: no S3 connector, narrow adopter base
- RIIIF: previous experience of poor performance (bundled with Samvera / Sufia)
- digilib: documentation and code are antiquated.
- iiif_s3: Level 0 implementation only.

2.3.4 Information Sources

This section contains information gathered from peer institutions who are implementing, or have implemented, the candidates previously listed.

Bodleian Libraries

Contact: Andrew Hankinson

The Bodleian Libraries at Oxford University use IIPImage. They serve about 1M images on 4 load-balanced servers with 4 IIPImage cores each.

Source images are lossless JPEG2000.

Bodleian has never had any problems with this setup outside of regular upgrade and maintenance tasks.

National Gallery of Art

Contact: David Beaudet

NGA uses IIPImage to serve Pyramidal TIFFs. They maintain a separate fork of IIPImage on Github where they made several improvements. The `nga_prod` branch is the one used in production at NGA. It is thoroughly tested and no problems have been reported so far.

There is an ongoing effort to add a S3 connector among others.

Topics on the NGA fork are occasionally merged into the upstream repo. There seems to be agreement that the S3 connector enhancement is important enough to deserve a pull request upstream. On the other hand, a solution that does not need implementing a S3 connector for IIPImage is available, as described in Access Master Cache.

The NGA branch also contains an enhancement that allows to properly upscale images via request headers. This is probably not a feature that we are interested in.

NGA went through an extensive data migration (100,000 images pulled from their legacy DAMS and converted to PTIFFs) which produced interesting results related to the PTIFF conversion. Vips was used to generate the pyramidal images which was many times faster than ImageMagick and produced images 1/3 of the size. A workaround was devised to overcome a scaling bug in VIPS. Details about the process are in the Recipes appendix.

Yale Center for British Art

Contacts: Michael Apppleby, Eric James

YCBA uses 2 instances of Cantaloupe pulling from S3, served by Cloudfront.

Performance has been quite stable and load is in the average very low. No major issues reported.

Wellcome Trust

Contact: Tom Crane, Digirati

Digirati has implemented a very large image set for the Wellcome trust (32M images) and is planning to more than double the amount (72M) soon.

This setup uses IIPImage with JPEG2000. The server is rock solid, but the main challenge is cache management for fast random access to derivatives.

The image server accesses a custom intermediate store which provides a NFS mount point. Derivatives most frequently accessed are stored there in faster SSD drives, least used ones are in S3. Performance tweaks, which are challenging given the size of the collection, are underway.

2.4 Gateway Service

In addition to implementing an image server, a "gateway" service needs to be implemented in order to capture and manipulate incoming requests. It is important that this component be both lightweight and modular, striving to tax the traffic as little as possible.

The Gateway Service needs to be able to fulfill different tasks (not all of them implemented from the outset):

- Providing a temporary mapping of IIIF Image API v3 features if the chosen image server does not support 3.0 at the moment of the launch.
- Map URLs beyond the capabilities of the image server.
- Implement caching strategies that the image server is not able to handle.
- Providing a redirect service for changing URIs.

Specific functions of the service are described below.

2.4.1 Image API versioning

While we want to adopt Image API v3 as soon as possible, it may be necessary to maintain compatibility with v2 for some time, maybe even for a long time if some live services depending on the v2 API are not updated. this is also valid for future major iterations of the IIIF API. Therefore, a versioning scheme should be devised.

One approach is to use a prefix to indicate which version of the API is being requested, e.g:

```
<webroot>/iiif/<API version>/<id>/<parameters>
```

Where `<API version>` is an integer corresponding to the major Image API version. A URL without `<API version/>` would redirect to the latest available version. This solution would work best with the Image API.

An alternative solution is to use content negotiation via HTTP headers. E.g.:

```
Accept: application/ld+json;profile=http://iiif.io/api/presentation/3/context.json
```

would request a V3 API. As for the URL-based approach, the default request would point to the latest version available on the server.

This approach is a better fit for the Presentation API.

This approach presents some problems with caching. Since different API versions have the same URL, a caching server such as Varnish cannot distinguish between the two. However there are solutions to this issue[7].

2.4.2 Caching

Raw image server performance represents only one of the factors for the successful implementation of an efficient and responsive image delivery system. The other major component is image caching.

In order to scale up to millions of images and potentially thousands of concurrent requests, a caching strategy must be devised. It is very unlikely that this strategy will be optimal from the outset, because there is no preexisting formula that can fit all the unique parameters that our system will present. However, having a starting setup that is as close as possible to the optimal one will be critical to a satisfactory user experience upon launch.

The factors upon which the caching strategy hinges off are:

- Serving both large and small images quickly (i.e. with the lowest possible latency perception for the end user).
- Avoiding astronomical storage costs.
- Avoiding duplicate processing.
- Avoiding undue manual maintenance.
- Being adaptable to adjustments driven by observing traffic patterns.

The strategy to be devised is based upon the following guidelines:

- Creating a source image cache that stores local copies of images retrieved from a network.
- Creating a derivative cache that stores generated derivatives so that they the image server doesn't have to process it for every request.
- Using fast, and presumably more expensive, storage (i.e. SSD) for the most frequently requestsd contents, and less expensive, less fast storage for the full data set, in a way that balances storage costs and availability. This is expected to be one of the most variable parameters that will need most adjustments over time.
- Set cache expiration policies based on age and/or checksum so that stale caches are automatically removed, without manual intervention needed. This also is a parameter that needs adjustments as experiecnce with real-world traffic is gathered.

[7] https://drive.google.com/open?id=1RL8cJSM1A4PSAgYyxRftatvtRQJK4XRK

Access Master Cache

This cache layer "lazily" (i.e. upon first user request) stores a copy of the access master in the most efficient storage tier possible (e.g. the image server's local filesystem) before sending the image data to the derivative processor. When another derivative for the same source is requested, if the cache has not expired, the local cache is used. Oherwise the source is retrieved and stored again.

It is important to clear the derivative cache when the source cache is cleared.

Cache Rotation

An additional feature is necessary for maintaining a reasonably sized cache, i.e. rotating out images from the cache according to some strategy. Since this layer is built on the fastest, and therefore most expensive, storage tier, we cannot afford to store a large amount of images that are rarely accessed in this tier.

In order to balance storage costs and performance, a script should be devised to prune the cache based on the files' *last access* timestamp.

Redirect Service

Even though image and presentation element identifiers should be as stable as possible, it is inevitable that with time some identifiers may become unavailable, even though the resource they refer to is still published. In order to ensure continuous availability of resources that are being pointed to by out of date Web document, an archive of historical identifiers with a reference to the most current, authoritative identifier should be maintained.

The Gateway Service is a convenient place to manage the redirection traffic by parsing incoming identifier requests, looking up the ID archive, and redirecting decommissioned ones to up to date ones or error pages (in case the resource is gone as well).

2.4.3 Derivative Cache

Similarly to the access master cache, the derivative cache is critical to a smooth user experience. The shim or the image server (depending on the server implemented should send a `304 NOT MODIFIED` status code if a derivative is requested that already exists in the derivative cache.

Derivative caching may be handled by the image server. Supposedly, the server creates and stores a new file for each specific combination of IIIF parameters and image identifier. What may be necessary to handle separately is pruning and purging. The former refers to periodically removing derivatives not recently used, using the same process as for the access masters; the latter, to removing all the derivatives of an access master when this is removed from the cache.

Thumbnails

A particular performance-related challenge is the creation of occasional thumbnails for large images. If an index page is requested containing many thumbnails of uncached images, those images must be retrieved in their master format to just create a thumbnail.

A possible solution to this issue could be to pre-compile thumbnails right after the access masters are generated. This way the processing effort is spread out in time instead of being concentrated at request time.

The issue with this approach is that there could be mutiple applications requesting different thumbnail sizes, and those sizes may change any time. It is not trivial to keep track of which thumbnail sets are to be maintained for all possible clients.

Another strategy could be to balance the cache expiration based on the image size; i.e. smaller images, whose deletion would reclaim little to negligible storage space, would be kept for longer time than larger images, whose cleanup is more effective.

The best initial approach may be to perform some usability tests before implementing any one-off caching strategy that creates more maintenance work. If performance results to be unacceptable for a large enough percentage of requests, such one-off solution may be necessary.

2.4.4 Web Front End & Load Balancing

In a production environment, it is imperative to run more than one image server instance in order to provide adequate bandwidth and service availability. These nodes need to be coordinated by a load balancer.

A very good candidate for this task is Nginx, which is a very robust, mature web server with a very small resource footprint. Nginx is able to handle large amounts of concurrent connections and has excellent load balancing capabilities.

For larger and more geographically distributed scenarios (as GCIS may very likely end up becoming), more complex setups may be orchestrated by dedicated services such as AWS CloudFront. This may also replace local caching proxies with its edge caching features. Nginx would still be used in front of each image server instance.

2.5 Manifest Service

Alongside with the image delivery (IIIF Image API) service and related tools, a manifest generation and delivery (IIIF Presentation) service needs to be implemented.

In broad lines, this service consists of an ETL framework that converts structural metadata related to the published images into IIIF manifests, and a web server that exposes such manifests to the Web.

The structure of such service is out of the scope of this document, since it is a component tightly related to the Getty's information architecture and data models.

As demand for more specific manifests representing special or user-defined collections of resources grows, a dynamic manifest generation service may become necessary. This service has not been completely defined in scope yet.

2.6 ETL And Migration

The access master images used by the image server to generate derivatives are created from specific files (called "Modified Masters" in Rosetta, and "Access Files" in OTMM) with specific characteristics (however the presence of outliers should be accounted for).

Master images are retrieved (for the first phase of this project) from two existing DAM servers: Rosetta (for research and library contents) and OTMM (for museum contents).

Two sets of discrete but related tasks need to be accomplished:

1. Retrieve all the images that make up the initial data set, and generate related manifests (migration).
2. Ensure that new images are automatically harvested and their related manifests are generated as they are created in the source repositories (ETL).

Both projects need ad-hoc code to be written which will use the same underlying business logic for connecting to the source systems, find and retrieve the images, convert them to access masters, and deposit them into a store accessible by the image server; as well as the manifest generation functionality.

2.6.1 ETL Framework

The ETL operations will be initially accessible via CLI. However, it is important to separate the interface layer from the controllers from the outset, so that a REST API can be easily added later on in case a remote system needs to interact with the framework (e.g. event-driven integration software). Initially, the most practical way to run the ETL operations is via cron jobs.

It is important to build a strong initial foundation for the ETL code, which should be easy to debug and expand. Configuration-based mapping, which allows to make minor changes to conversion parameters without touching the core business logic, is strongly recommended.

2.6.2 Migration Scripts

The part of the code responsible for one-time migration(s) will be eventually discarded, however it is important to build it so that it can be repurposed for related projects later. These scripts should be able to import the core functionality of the ETL framework as a module in order to reuse common business logic.

2.7 Storage Technology and Vendors

Since storage is going to be a major cost factor for this project, it may be valuable to shop for different service providers offering storage that uses the S3 protocol. There are a number of such services[8], and while Amazon is the most mature of all and offers competitive prices, a comparison may be still worth while (unless there is a specific reason to consistently use AWS).

3 Benchmarks

The purpose of these benchmarks is to establish the most resource-efficient combination of server software, image processor, and source image format.

Note: This is not to be considered the *only* factor to pick a winner. Other important factors should be considered as indicated in the Key Requirements.

[8] https://en.wikipedia.org/wiki/Amazon_S3#S3_API_and_competing_services

3.1 Image Formats & Encoding

3.1.1 Test Sources and Setup

About 1,000 sample images have been chosen from the GM and GRI archives to provide an initial data set to test the encoding and decoding of PTIFF and JP2 from high-resolution TIFF images.

The images have been encoded to preserve original size, aspect ratio and cropping area. All images have been converted to a sRGB color profile if they had a different color profile assigned.

PTIFFs used for this benchmark have been encoded with the previously described tools. JP2s have been encoded with Kakadu using the following script:

```
kdu_compress -i $in_path -o $out_path Clevels=6 Clayers=6 \
"Cprecincts={256,256},{256,256},{128,128}" "Stiles={512,512}" Corder=RPCL \
ORGgen_plt=yes ORGtparts=R "Cblk={64,64}" -jp2_space sRGB Cuse_sop=yes \
Cuse_eph=yes -flush_period 1024 -rate 3
```

Of the 1,000 sample images, 6 of them have been analyzed visually, comparing each source TIFF with its PTIFF and JP2 derivatives.

3.1.2 Test Results

Encoding Speed

The time elapsed to convert all the Museum images (504 files, ~44 Gb) has been used to measure the encoding performance by using the `time` command on a batch conversion script.

Time of conversion for PTIFF:

```
real    79m28.083s
user    14m49.999s
sys     21m54.979s
```

Time of conversion to JP2:

```
real    25m31.415s
user    29m3.426s
sys     7m36.416s
```

PTIFF encoding took over 3 times as long as JP2 encoding.

One thing to be noted is that all along the conversion process, the PTIFF conversion script ran single-threaded, while Kakadu ran across all 4 processors available.

Decoding Speed

Decoding speed has been measured through IIIF server delivery. IIPImage is the only server tested that supports both PTIFF and JP2, and both setups have been tested. The results are in the Image Server Benchmark section.

Image Size

Image size comparison is as simple as running du on the folders containing the respective PTIFF anmd JP2 derivatives.

PTIFF dfolder size:

```
10412528 ptiff
```

JP2 dfolder size:

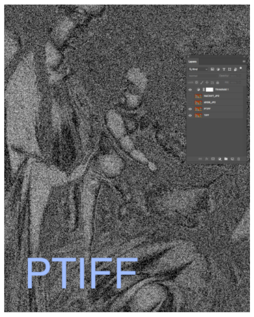
```
11259444 jp2
```

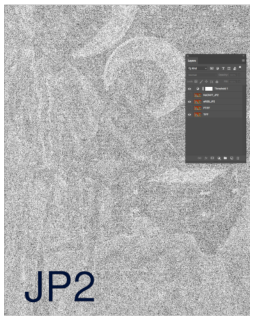
The JP2 folder overall is about 8% larger. Some JP2 images are slightly smaller, some others slightly larger.

Image Quality

The quality of the PTIFF and JP2 derivatives was compared in terms of color fidelity to the original image. The Imaging department ran some analysis of the derivative images that highlights the pixel-wise value differences from the original in each image.

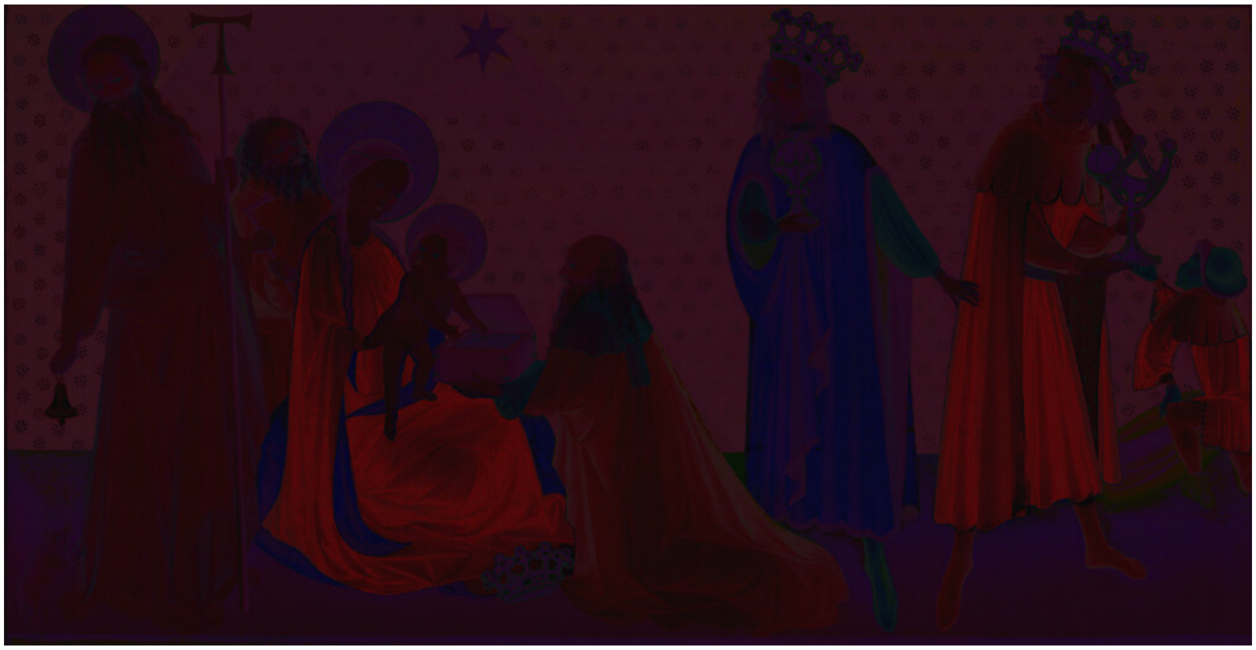
The following image shows the resulting analysis. On the left of the image is the detail of the original TIFF image sampled. On the top and bottom right corner are measurements of deviation from the original of, respectively, the PTIFF and JP2 files. Black pixels indicate an exact match with the original value, white a non-match.



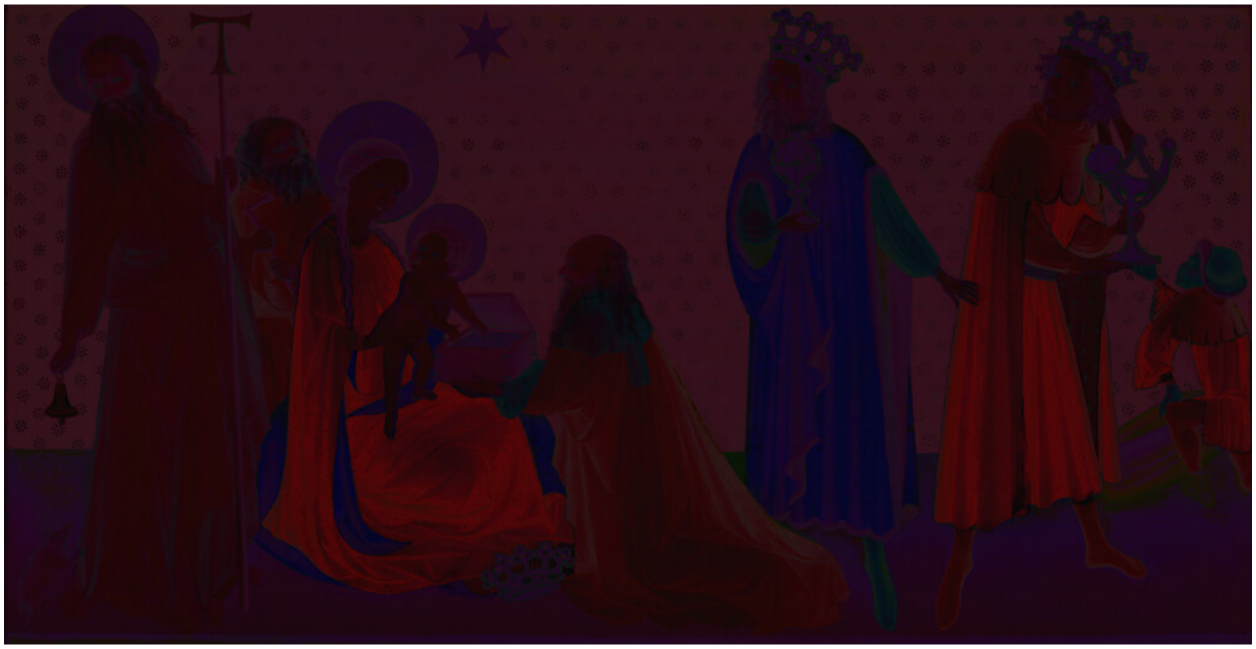


Note that the *amount* of deviation is not measured in the image above. To the naked eye, the images look nearly identical. In fact, a quantitative analysis of the different pixels reveals no clear difference:

Original image



Quantitative difference (brighter areas are more different) between original and PTIFF derivative.



Quantitative difference (brighter areas are more different) between original and JP2 derivative.

Note: the two delta images have been equally gamma-corrected for readability.

3.2 Image Servers

3.2.1 Reference Data Set

The reference data set used for the benchmark is derived from 1,000 distinct image files, semi-manually chosen to represent wide diversity in size, color distribution, embedded metadata, original format, etc.

Images are coarsely grouped by size ranges, in order to make it easier to perform tests based on large vs. small sources.

2 sets of source files are derived from the above 1,000 images: one as JP2 and one as pyramidal TIFF.

Note: Given the heterogeneous sources of the production masters, variations in all the parameters mentioned below should be expected in the source images.

Image Encoding Parameters

- Original size
- RGB, 8 bit per channel
- All layers flattened
- Non-RGB channels discarded
- sRGB Color profile
- All EXIF metadata retained

For JPEG2000:

- Compression: lossy (JPEG), quality: 90%
- Resolutions: variable, based on image size
- Precinct size: 256x256
- Code block size: 64x64

For Pyramidal TIFF:

- Pixel order: tiled
- Compression: lossy (JPEG), quality: 90%
- Resolutions: variable, based on image size
- Tile size: 256x256

3.2.2 Load Test Servers

Each IIIF server setup consists of two or more Docker containers, one of which contains an Nginx server and load balancer, and the other(s) one or more instances of the image server put to the test.

All the setups are deployed an run within their containers inside an AWS EC2 instance (the test "server") with the following characteristics:

- CPU: 8x Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz
- RAM: 32 Gb
- Data storage: Amazon gp2 type (SSD)

In all cases, all caches are turned off, i.e. each request sent to the server results in a full computation of a derivative.

A separate host is dedicated to the client that performs the load test (the test client):

- CPU: 4x Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz
- RAM: 16 Gb
- Data storage: Amazon gp2 type (SSD)

3.2.3 Benchmark Parameters

The benchmark consists in comparing the finalist products based on different "axes": server setup, image sizes, and concurrent requests. A test is performed for each of the variants in each axis, if compatible (e.g. Cantaloupe and Loris do not support PTIFFs so the test with PTIFFs is skipped for those servers).

Axis 1: Server Setup

The following server setups are tested:

Cantaloupe with Kakadu

Cantaloupe runs in a single container and automatically spawns processes as needed. Heap size is statically set to 12Gb.

Note: Cantaloupe with PTIFF is not tested. This is because for optimal TIFF performance, the `JaiProcessor` should in theory be used according to the manual. However, the same site reports that JAI has been long deprecated. The native Java2D processor can be used, but it is considerably slower. For this reason, it is probably not worth testing Cantaloupe with PTIFF images.

IIPImage with Kakadu

The standalone `iipsrv.fcgi` FastCGI aplication does not spawn sub-processes automatically, therefore 6 separate containers are deployed, each one with a single IIPImage instance. Nginx is set up to act as a load balancer.

It is mentioned in the IIPImage documentation that OpenJPEG support is in the works, but there is no timeline for its release, therefore IIPImage with OenJPEG is not tested.

IIPImage with PTIFF

This is a version of IIPImage without Kakadu support. This removes a number of system dependencies. The base image for this build is Alpine Linux, while the builds including Kakadu are ArchLinux. This setup as well consists of 6 IIPImage containers load balanced by Nginx.

Loris with Kakadu

Loris server is obtained by a stock image with the configuration file modified to fit within the benchmark parameters. For some reason, the caching directives in the passed configuration did not seem to be honored, and the logging showed that caching was *not* disabled completely. It is uncertain at the moment which implications on real performance this would have.

6 Loris cores are deployed in this setup, load balanced by Nginx.

Axis 2: Source & Derivative Sizes

Each of the server setups is tested with ranges of source image sizes, if an organization of these by size ranges is possible. Different tests with different derivative sizes are also performed:

- Large sources (>75 Mpx), full size derivatives
- Large sources, large derivatives (3000px on longer side)
- Large sources, thumbnails (125px on longer side)
- Medium sources (between 10 and 75 Mpx), full size derivatives
- Medium sources, large derivatives
- Medium sources, thumbnails
- Small sources (< 10 Mpx), full size derivatives
- Small sources, large derivatives
- Small sources, thumbnails

Axis 3: Concurrent Connections

Each of the server setup + source/derivative combinations is tested with different sets of concurrent users to benchmark parallelism:

- 10 concurrent connections
- 100 concurrent connections
- 1,000 concurrent connections

Note: 1000 connections is an extremely large number, that could be reached in a real world scenario only very rarely and for very brief moments. In production, it is assumed that the vast majority of the requests is fulfilled by the cache, while the test removes the cache completely.

3.2.4 Test Instruments and Methodology

The test is carried out using Locust, a Python-based HTTP load testing application whilch allows a great deal of flexibility in automating requests.

Each load test session targets one server setup and performs all combinations of requests within a fixed time of 10 minutes. Some parameters are randomized, others are fixed.

Each Locust client performs the following actions:

- Select an image identifier from one of three lists. One list contains identifiers of images of less than 10 Mpx, one images between 10 and 75 Mpx, and one 75 Mpx and up.
- For each image identifier, the client requests:
  - 50 thumbnails of 128 pixels on the longer side;
  - 10 images of 3000 pixels on the longer side;
  - 1 image at native size.

During the load test Locust is run in "no UI" mode that dumps results in a CSV file. These results, attached in Appendix 2, are broken down by combination of source image size and derivative size.

All servers ran with all caches turned off (except for Loris, as noted above).

3.2.5 Test Result Summary

Output

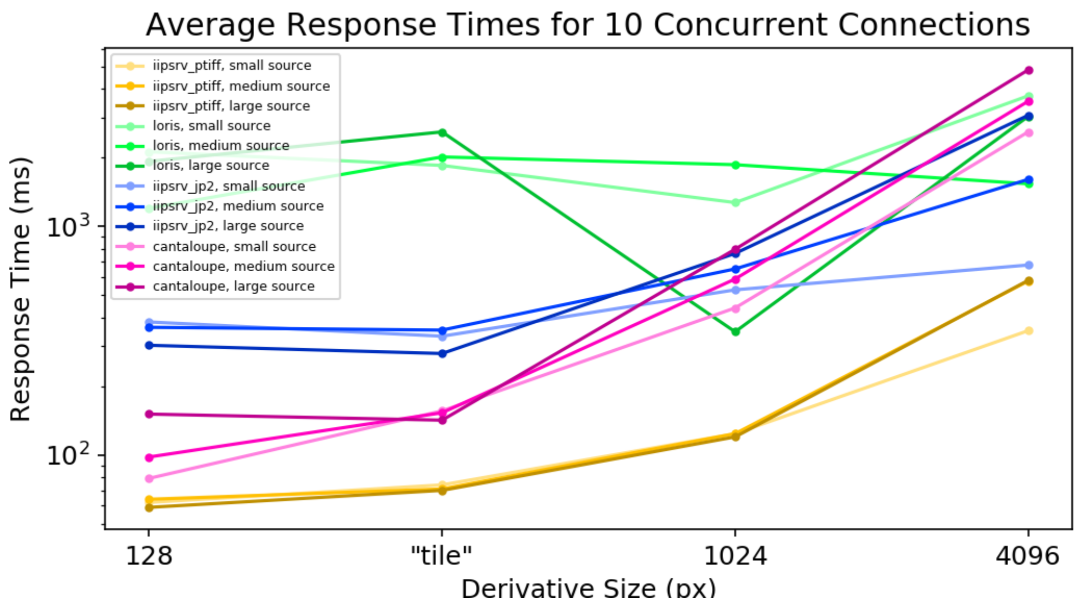
The fastest and most consistent server setup is IIPImage with pyramidal TIFF images. A similar output consistency is provided by IIPImage with Kakadu, albeit with a lower rate of requests per second.

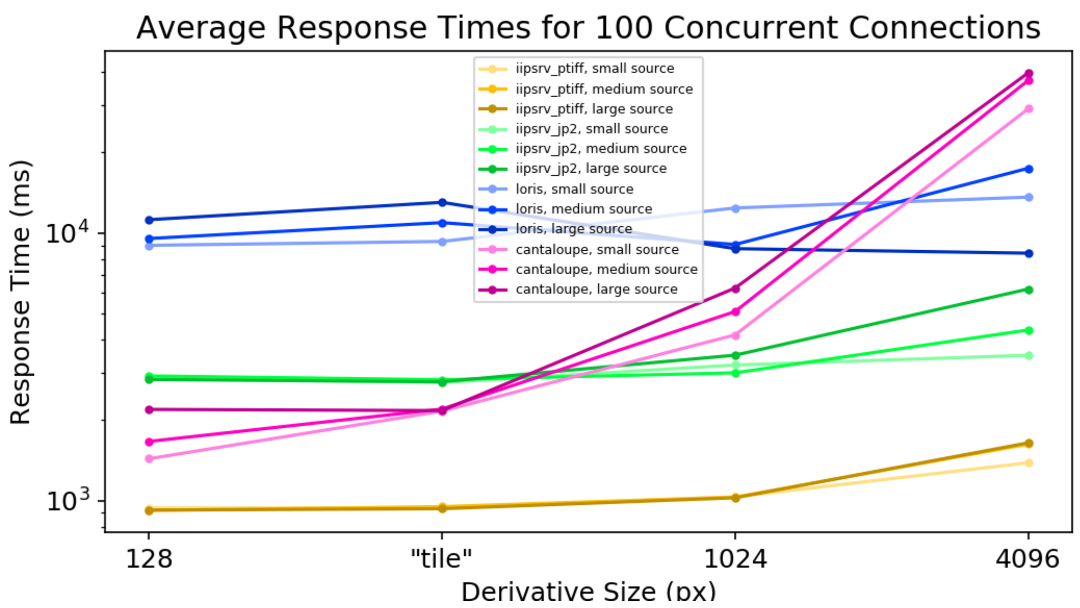
Cantaloupe maintains a quite good performance, but still quite distanced from the IIPImage results.

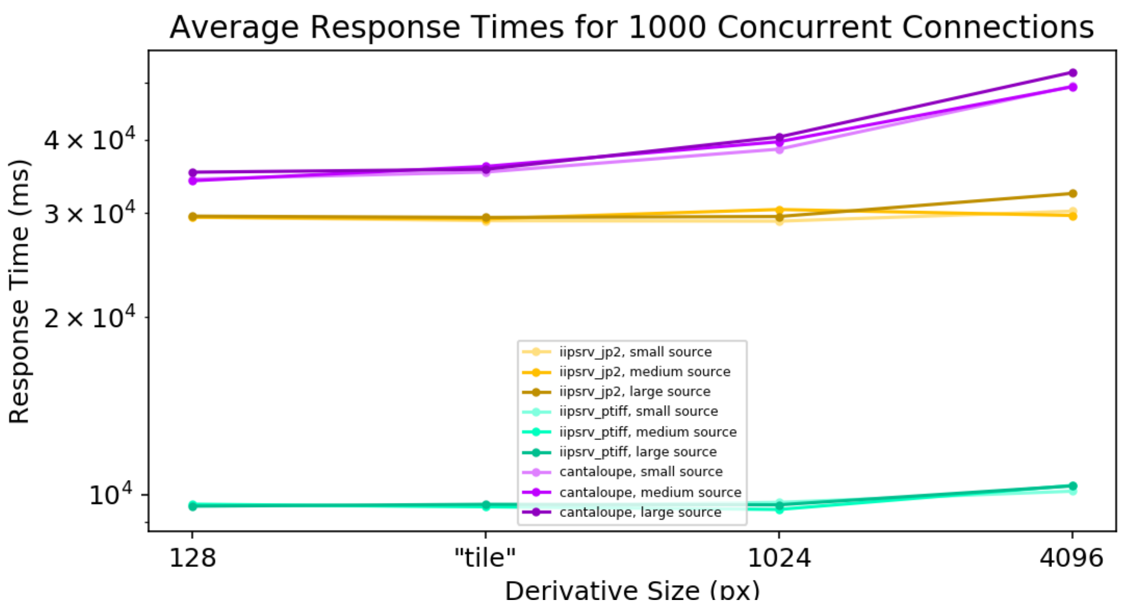
Loris has the lowest output rate by far. It is also extremely uneven in output rate and resource usage. In all benchmarks, and in particular the one with 10 concurrent connections, there are wide gaps in which the output is near zero and the resource usage seems to plummet for several seconds. It is also noteworthy that only 3 of the 6 available cores were sending any output.

Average Response Times

Given the very broad distribution of response times, especially at 100 concurrent connections and above, it is not easy to distinguish the bottleneck caused by the job queue and the one from the actual image processing. The latter is probably most prominent in the 10 connections range.



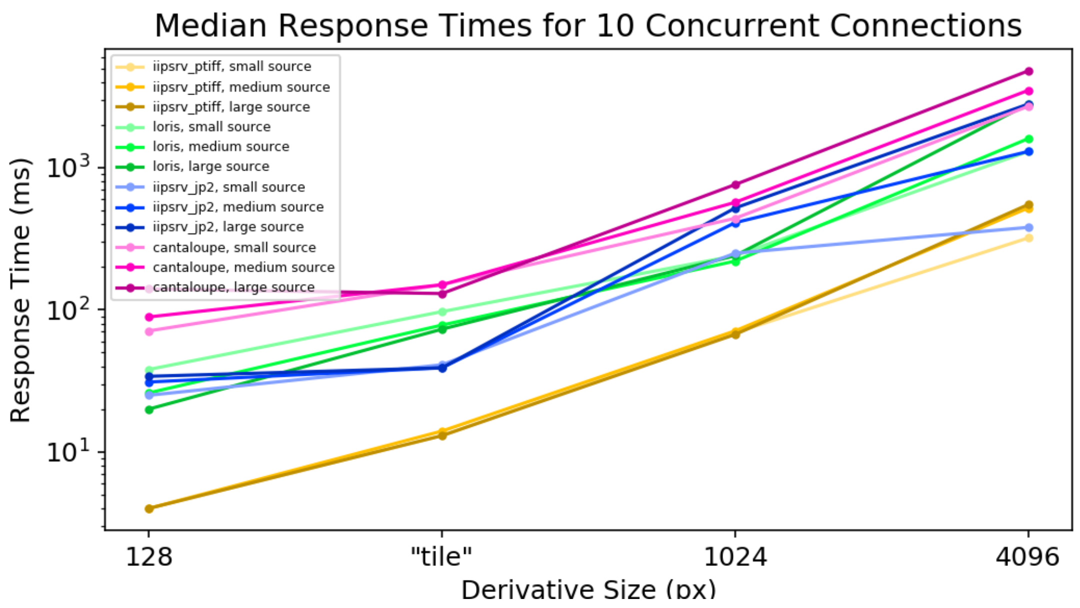


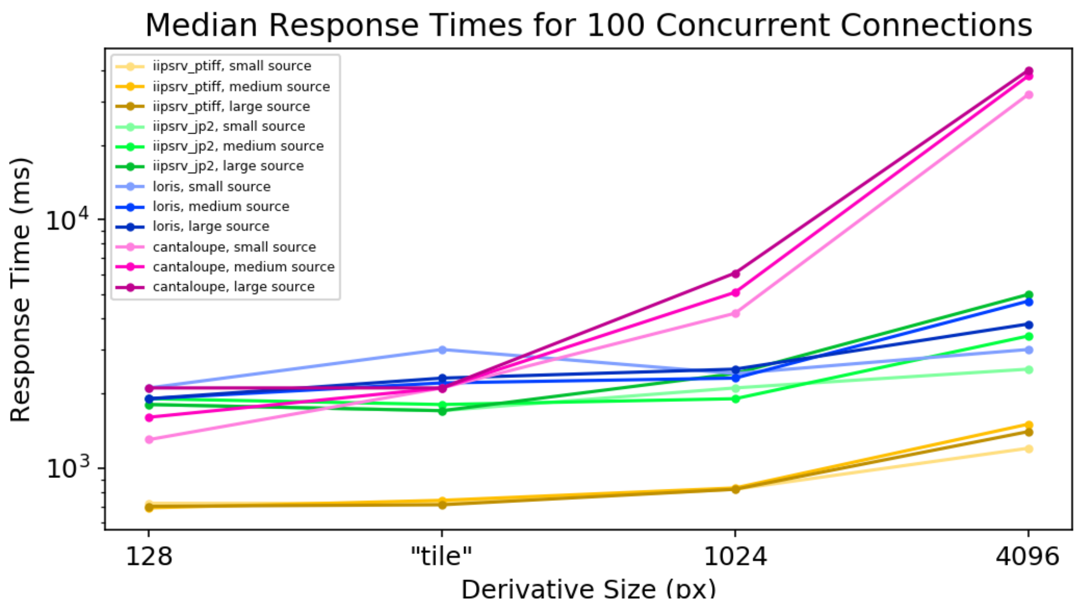


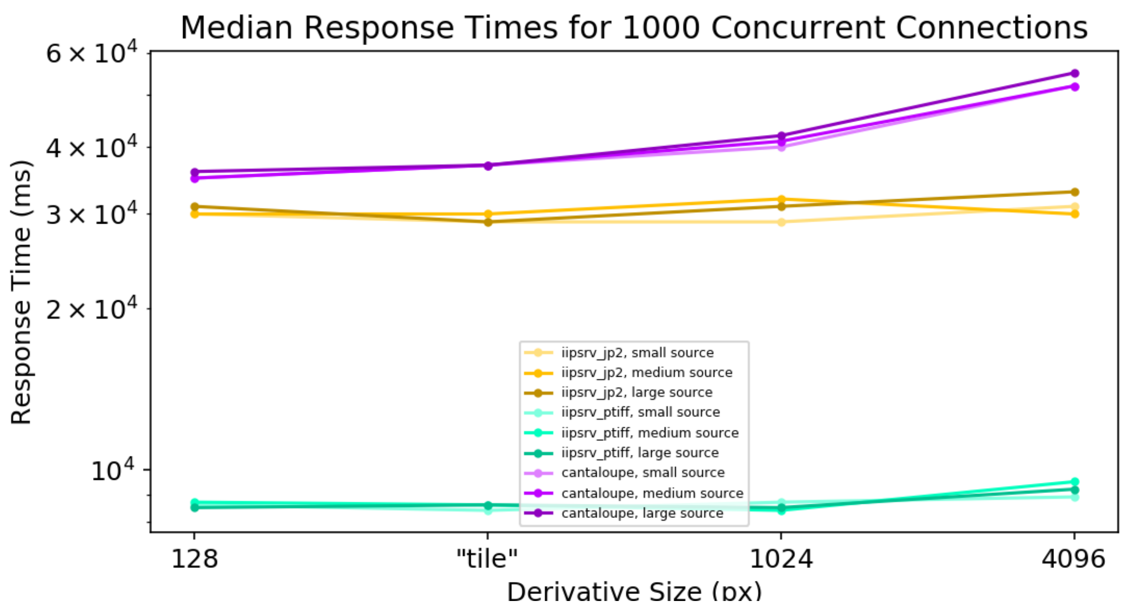
Note that some results are not quite consistent. Probably taking a larger sample by prolonging the test time and/or having different derivative size brackets (e.g. 128, 1024, 3000, full) may yield more reliable results.

Median Response Times

Given that about 50% of the response times are in the 50% percentile, it may be worth while considering the median measurements as well:



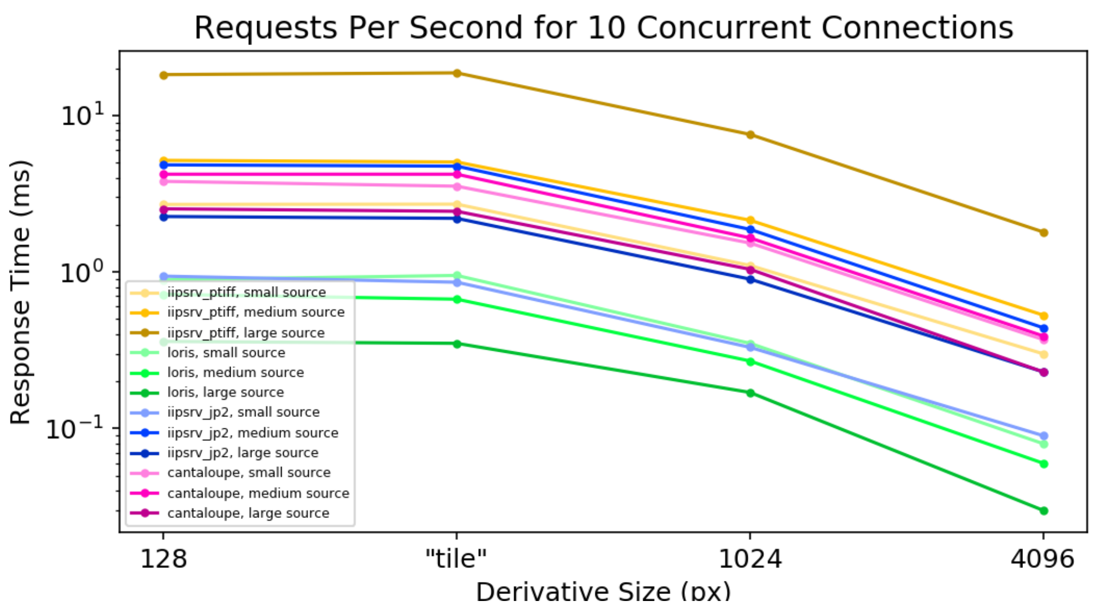


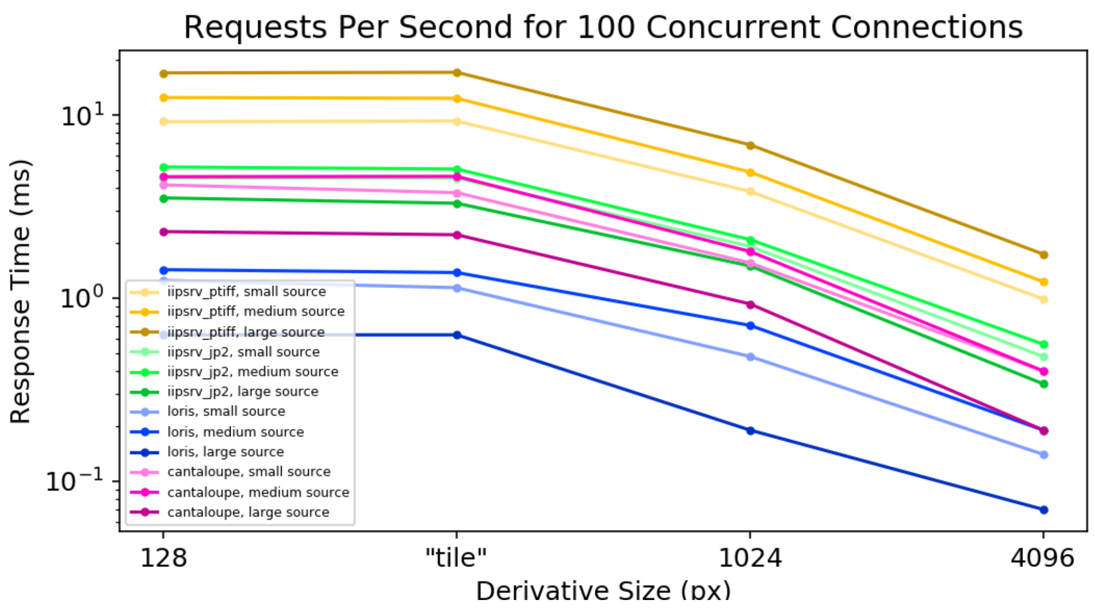


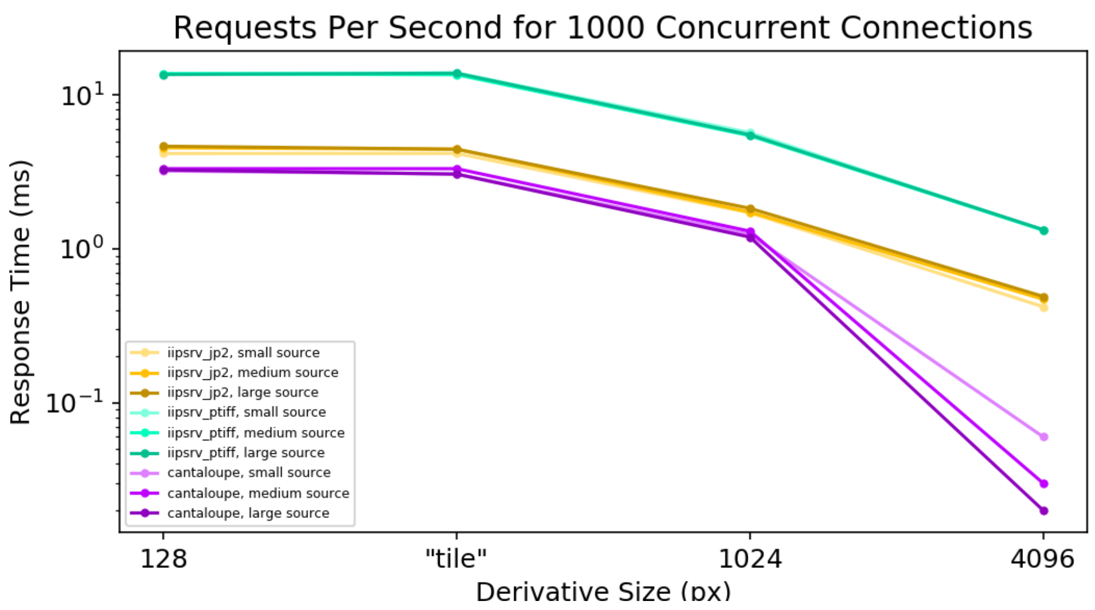
Note that Loris, when it does not return an error code, is actually quite competitive. It has not been discerned, however, whether the low times are due to caching, which was apparently not completely disabled.

Requests Per Second

The RPS should increase with the number of concurrent connections.


Requests Per Second for 10 Concurrent Connections
Response Time (ms)
10^1
10^0
10^{-1}
iipsrv_ptiff, small source
iipsrv_ptiff, medium source
iipsrv_ptiff, large source
loris, small source
loris, medium source
loris, large source
iipsrv_jp2, small source
iipsrv_jp2, medium source
iipsrv_jp2, large source
cantaloupe, small source
cantaloupe, medium source
cantaloupe, large source
128
"tile"
1024
4096
Derivative Size (px)


Requests Per Second for 100 Concurrent Connections
Response Time (ms)
10^1
10^0
10^{-1}
iipsrv_ptiff, small source
iipsrv_ptiff, medium source
iipsrv_ptiff, large source
iipsrv_jp2, small source
iipsrv_jp2, medium source
iipsrv_jp2, large source
loris, small source
loris, medium source
loris, large source
cantaloupe, small source
cantaloupe, medium source
cantaloupe, large source
128
"tile"
1024
4096
Derivative Size (px)



Stability

IIPImage yielded no failed request in 10 minmutes of full activity, in all benchmarks. Some 404 errors in the JP2 results were caused by a missing image.

Cantaloupe has a notable error rate—mostly gateway timeouts, but Java errors have been noticed in the application logs as well. It is likely that adjusting memory settings would resolve at least some of those issues.

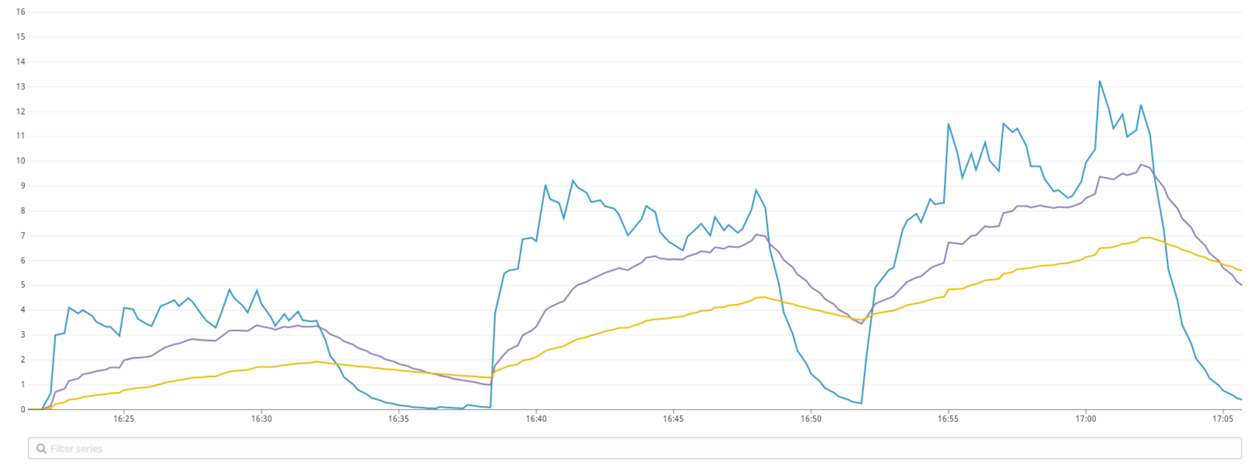
With Loris, 10 concurrent connections returned a conspicuous number of errors. 100 concurrent connections resulted in an overwhelming majority of failures (about 42 errors for each successful request). A test with 1000 connections was not performed.

Test result details, including request charts, are attached in Appendix 2.

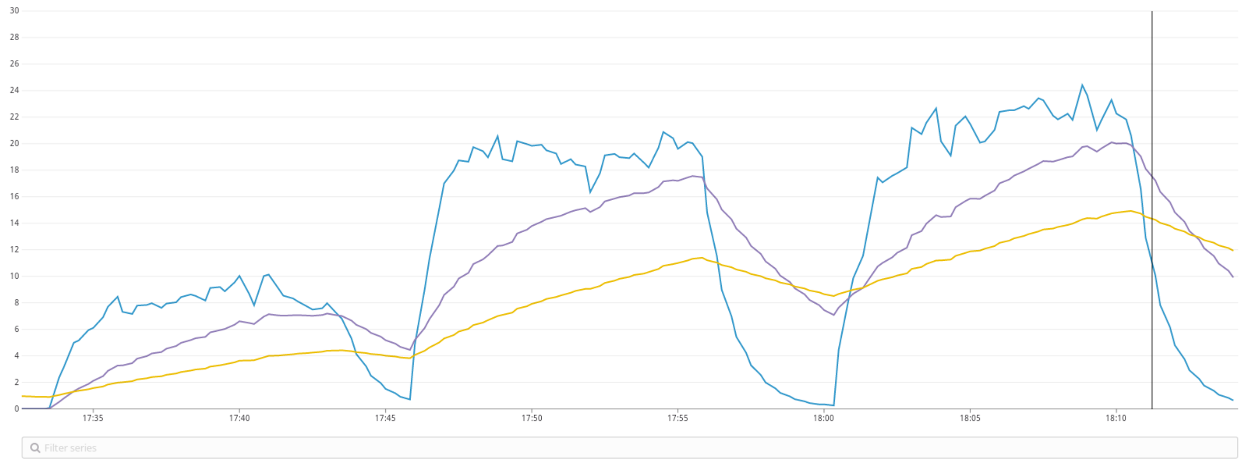
Resource Usage

The following images illustrate resource usages for each of the test sets with 10, 100 and (where applicable) 1000 concurrent connections.

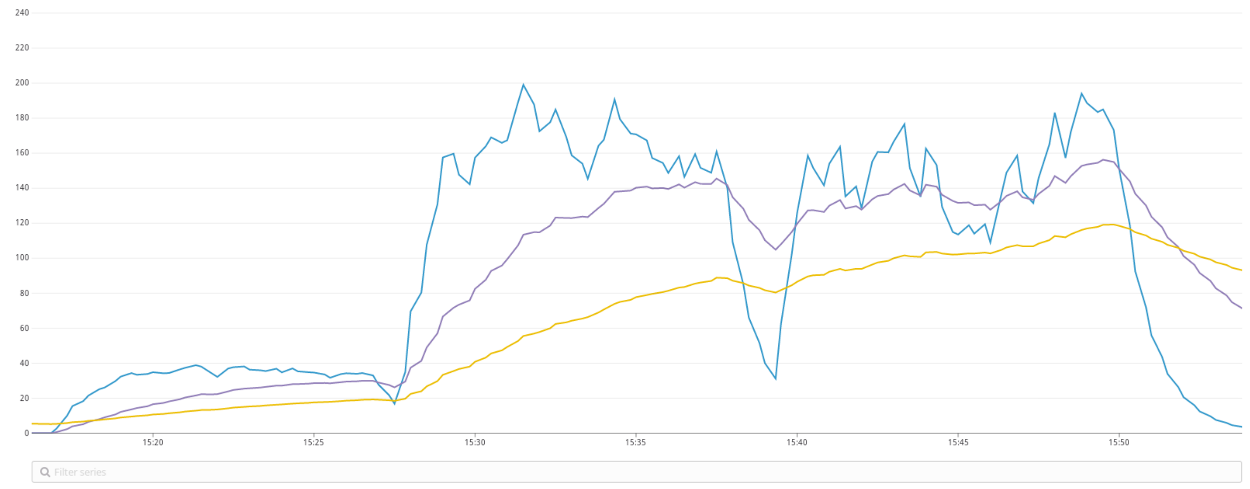
Average Load



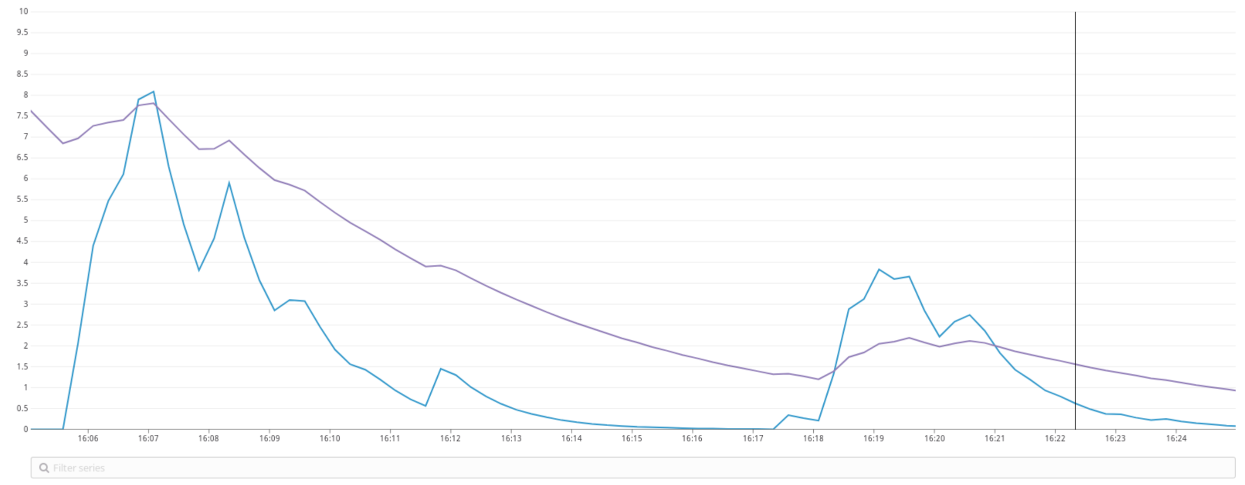
	Value	Min	Avg	Max	Metric	Tags ↓
🟦	0.46	0	5.21	13.25	system.load.1	host:or-dev-ec2-iiif-ubuntu-01
🟨	5.65	0	3.26	6.93	system.load.15	host:or-dev-ec2-iiif-ubuntu-01
🟪	5.15	0	4.64	9.87	system.load.5	host:or-dev-ec2-iiif-ubuntu-01



	Value	Min	Avg	Max	Metric	Tags ↓
🟦	0.64	0	12.11	24.41	system.load.1	host:or-dev-ec2-iiif-ubuntu-01
🟨	11.93	0.9	8.11	14.93	system.load.15	host:or-dev-ec2-iiif-ubuntu-01
🟪	9.91	0.01	11	20.1	system.load.5	host:or-dev-ec2-iiif-ubuntu-01



	Value	Min	Avg	Max	Metric	Tags ↓
🟦	4.62	0	99.26	199.07	system.load.1	host:or-dev-ec2-iiif-ubuntu-01
🟨	94.58	5.26	62.99	119.23	system.load.15	host:or-dev-ec2-iiif-ubuntu-01
🟪	74.89	0.07	89.74	156.27	system.load.5	host:or-dev-ec2-iiif-ubuntu-01

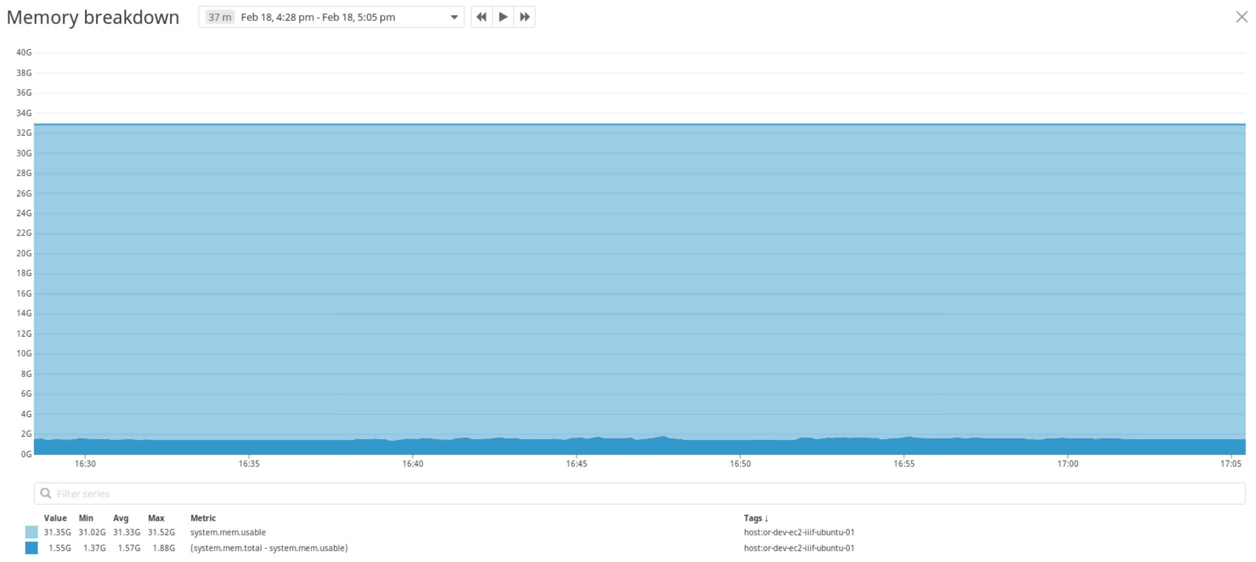


	Value	Min	Avg	Max	Metric	Tags ↓
🟦	0.09	0	1.7	8.09	system.load.1	host:or-dev-ec2-iiif-ubuntu-01
☐	13.58	13.58	26.54	45.12	system.load.15	host:or-dev-ec2-iiif-ubuntu-01
🟪	0.96	0.96	3.41	7.81	system.load.5	host:or-dev-ec2-iiif-ubuntu-01

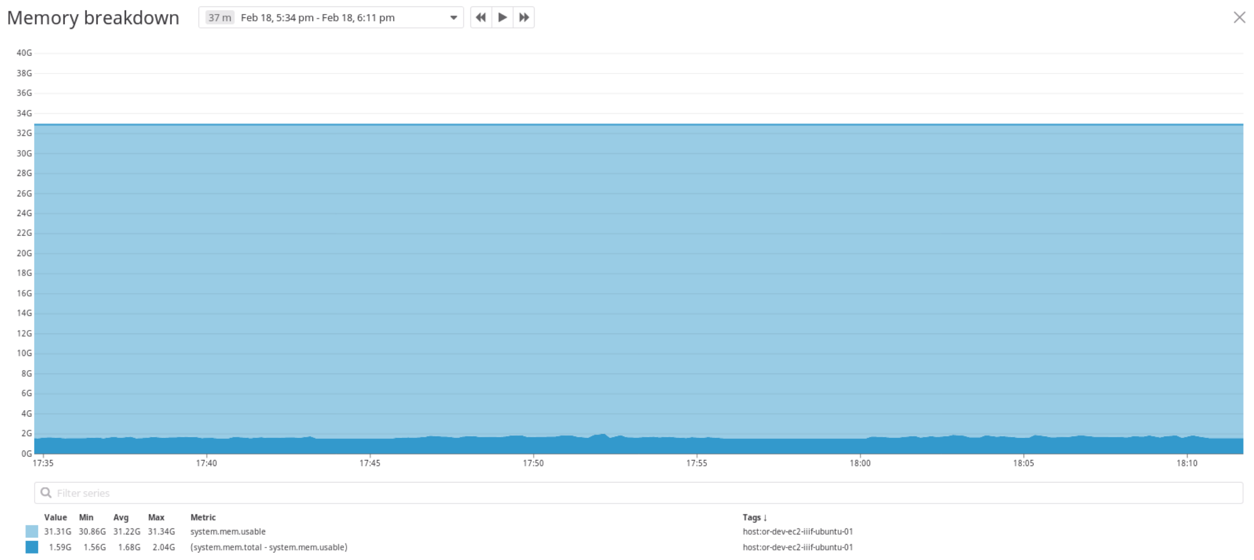
Memory Usage

The following graphs show memory usage of the whole system. These charts may be interesting for the purpose of horizontal scalability, where the allocated memory of dynamically generated processes is a major cost factor.

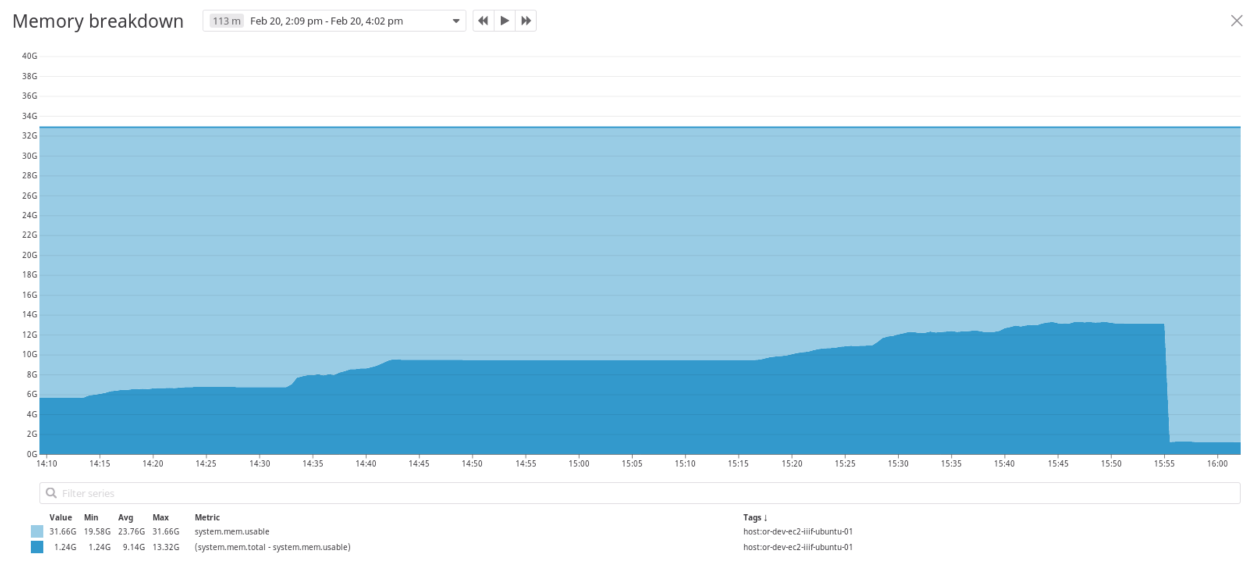
The only setup that displays a remarkable memory consumption is Cantaloupe, as expected from a high-load Java application.



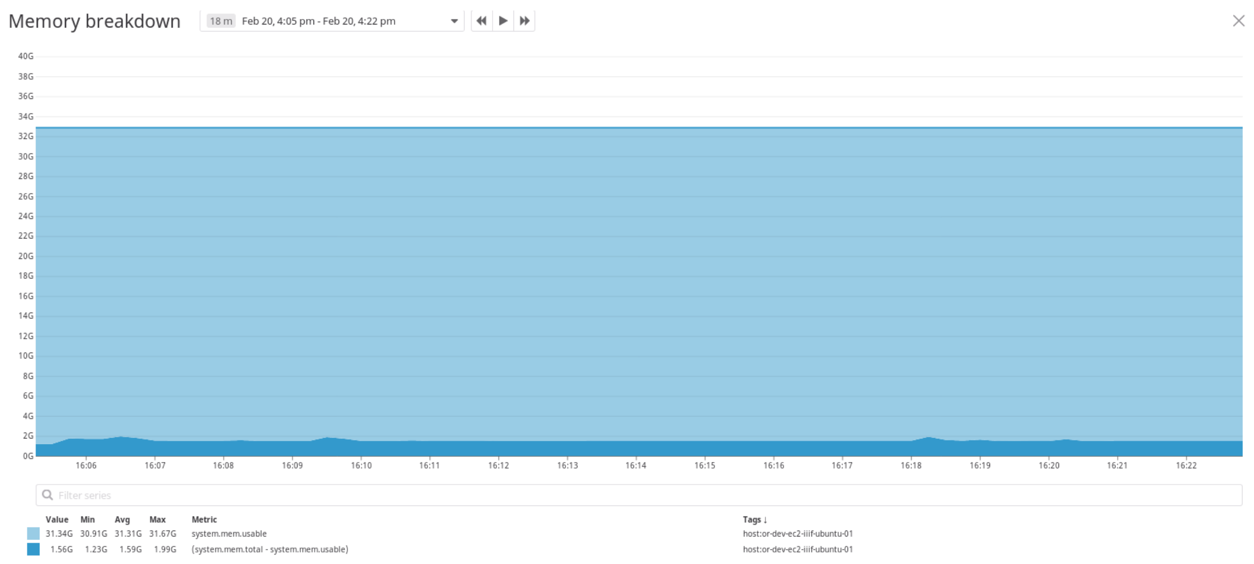
	Value	Min	Avg	Max	Metric	Tags ↓
🟦	31.35G	31.02G	31.33G	31.52G	system.mem.usable	host:or-dev-ec2-iiif-ubuntu-01
🟦	1.55G	1.37G	1.57G	1.88G	(system.mem.total - system.mem.usable)	host:or-dev-ec2-iiif-ubuntu-01



	Value	Min	Avg	Max	Metric	Tags ↓
🟦	31.31G	30.86G	31.22G	31.34G	system.mem.usable	host:or-dev-ec2-iiif-ubuntu-01
🟦	1.59G	1.56G	1.68G	2.04G	(system.mem.total - system.mem.usable)	host:or-dev-ec2-iiif-ubuntu-01



	Value	Min	Avg	Max	Metric	Tags ↓
🟦	31.66G	19.58G	23.76G	31.66G	system.mem.usable	host:or-dev-ec2-iiif-ubuntu-01
🟦	1.24G	1.24G	9.14G	13.32G	(system.mem.total - system.mem.usable)	host:or-dev-ec2-iiif-ubuntu-01



	Value	Min	Avg	Max	Metric	Tags ↓
🟦	31.34G	30.91G	31.31G	31.67G	system.mem.usable	host:or-dev-ec2-iiif-ubuntu-01
🟦	1.56G	1.23G	1.59G	1.99G	(system.mem.total - system.mem.usable)	host:or-dev-ec2-iiif-ubuntu-01

4 Conclusions

IIPImage is clearly the most suitable server for the GCIS project, given the following advantages:

- Superior performance
- Highest consistency of results

- Highest stability
- High scalability
- Resource efficiency
- Widespread adoption
- Longest history
- Solid development support

It is woth noting how intense traffic can put even IIPImage in a difficult spot. Even though IIPImage never failed to deliver an image during the tests, most of the timings in the 100-connection scenario (which is quite plausible) are unacceptable. This will have to be resolved with a judicious scaling setup or by further configuration adjustments for the application, load balancer and system.

Pyramidal TIFF (PTIFF) is the file format to be adopted for the access masters, given the following advantages over JPEG2000 (JP2):

- Much faster decoding speed
- No need to depend on commercial software
- More processing options

The only clear drawback of PTIFF is the encoding time, as mentioned previously. Other advantages of JP2 may be related to partial retrieval of the images, such as in tile extraction, but that has not yet been framed into a practical use case. The performance benchmarks include random region requests that seem to perform very well.

These conclusions should guide the next implementation steps, consisting of setting up an IIPImage cluster for production use and converting source images into access masters. Very careful testing, in a wider variety of scenarios than the ones used for the benchmarks presented here, should be performed in order to determine the optimal setup. Adjustments and compromises will very likely have to be made, e.g. to favor faster delivery of frequently requested sizes such as thumbnails, while avoiding exceedingly long delays for unusually large derivative and/or unusually high load.

Scaling strategies should be carefully studied as well. If using AWS autoscaling, it is necessary to understand how each IIPImage instance uses its own resources and how it performs parallel work with the other instances.

5 Appendix 1: Recipes

5.1 JP2 encoding parameters

The following are command-line parameters used by different implementers to encode lossy JPEG2000 files.

The Getty (thanks to Chris Edwards):

```
kdu_compress -i input.tif -o output.jp2 \
-rate 1.5 Creversible=yes Clayers=1 Clevels=7 \
Cprecincts={256,256},{256,256},{128,128} Corder=RPCL \
ORGgen_plt=yes ORGtparts=R Cblk={64,64}  Cuse_sop=yes
```

From the image_processing Wiki (with all options explained):

```
kdu_compress -i input.tif -o output.jp2 Clevels=6 Clayers=6 \
"Cprecincts={256,256},{256,256},{128,128}" "Stiles={512,512}" \
Corder=RPCL ORGgen_plt=yes ORGtparts=R "Cblk={64,64}" \
Cuse_sop=yes Cuse_eph=yes-flush_period 1024 -rate 3
```

The following are gathered from a recent survey on IIIF implementations. There is no association to the institution using these. :

```
kdu_compress -i <path> -o <path> -rate - Creversible=yes Clevels=6 \
Clayers=6 Cprecincts={256,256},{256,256},{128,128} Stiles={512,512} \
Corder=RPCL ORGgen_plt=yes ORGtparts=R Cblk={64,64} Cuse_sop=yes \
Cuse_eph=yes -flush_period 1024
```

5.2 Source Image Generation

Very efficient generation of PTIFF files was performed by NGA with `vips` and `tiffcp` to get around a VIPS bug[9]:

https://github.com/NationalGalleryOfArt/iipsrv/tree/master/imagescripts

This script has been later converted into Python for the production ETL setup. This code has not yet been released to the public as it is part of the larger GCIS ETL framework.

[9] LONG thread between D. Beaudet and the VIPS maintainer: https://github.com/libvips/libvips/issues/659

6 Appendix 2: Locust Test Data

The following tables are raw data dumps from the Locust load testing sessions. The column names have been abbreviated for compactness as follows:

- `src min`: Minimum size (in megapixels) of random source image selected.
- `src max`: Maximum size (in megapixels) of random source image selected.
- `derv sz`: Derivative size in its longer dimension.
- `# req`: Number of total requests completed successfully within the 10 minute time frame of the test.
- `# fail`: Number of failed requests (`4xx` or `5xx` HTTP codes).
- `median`: Median response time.
- `avg`: Average response time.
- `min`: Miminum response time.
- `max`: Maximum response time.
- `avg sz`: Average response size.
- `req/s`: Requests per second.

All timings are in milliseconds.

6.1 IIPImage With PTIFF

6.1.1 10 Connections

’src min’	’src max’	’derv sz’	’# req’	’# fail’	’median’	’avg’	’min’	’max’	’avg sz’	’req/s’
0	10	tile	1597	0	13	74	2	1418	42960	2.71
0	10	1024	646	0	69	124	15	1269	129884	1.10
0	10	128	1594	0	4	62	2	1321	3512	2.70
0	10	4096	174	0	320	350	76	1100	1373086	0.30
10	75	tile	2974	0	14	71	7	968	48083	5.04
10	75	1024	1261	0	71	124	10	1080	149341	2.14
10	75	128	3044	0	4	64	2	1397	3976	5.16
10	75	4096	310	0	520	577	124	1436	2211692	0.53

’src min’	’src max’	’derv sz’	’# req’	’# fail’	’median’	’avg’	’min’	’max’	’avg sz’	’req/s’
75	-	tile	11020	0	13	70	6	1296	35544	18.69
75	-	1024	4460	0	67	120	16	1030	162239	7.56
75	-	128	10744	0	4	59	2	1297	4011	18.22
75	-	4096	1059	0	550	581	170	1470	2375629	1.80
’any’	’any’	’any’	38883	0	13	94	2	1470	131169	65.94

6.1.2 100 Connections

’src min’	’src max’	’derv sz’	’# req’	’# fail’	’median’	’avg’	’min’	’max’	’avg sz’	’req/s’
0	10	tile	5474	0	720	940	2	4588	46697	9.29
0	10	1024	2258	0	820	1033	21	4712	130108	3.83
0	10	128	5424	0	720	940	2	4702	3652	9.21
0	10	4096	582	0	1200	1386	130	4745	1439533	0.99
10	75	tile	7283	0	740	950	7	5031	47797	12.37
10	75	1024	2883	0	830	1031	22	4587	141058	4.89
10	75	128	7355	0	690	920	2	5187	3723	12.49
10	75	4096	723	0	1500	1626	271	5261	2178882	1.23
75	-	tile	10102	0	710	933	7	5135	35709	17.15
75	-	1024	4061	0	820	1025	25	4743	159772	6.89
75	-	128	10030	0	700	922	2	5199	3973	17.03
75	-	4096	1023	0	1400	1645	319	4624	2280863	1.74
’any’	’any’	’any’	57198	0	770	975	2	5261	124956	97.11

6.1.3 1000 Connections

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
0	10	tile	8172	0	8400	9551	3	27795	44181	13.87
0	10	1024	3341	0	8700	9700	32	27796	121848	5.67
0	10	128	8100	0	8600	9586	3	27711	3555	13.75
0	10	4096	778	0	8900	10131	180	27579	1394046	1.32
10	75	tile	7947	0	8600	9534	9	27914	48207	13.49
10	75	1024	3195	0	8400	9435	41	27931	147296	5.42
10	75	128	8076	0	8700	9629	2	27644	3814	13.71
10	75	4096	775	0	9500	10361	180	27399	2269113	1.32
75	-	tile	8124	0	8600	9625	8	27784	33832	13.79
75	-	1024	3231	0	8500	9606	40	27845	159088	5.49
75	-	128	7968	0	8500	9557	2	27901	3970	13.53
75	-	4096	781	0	9200	10343	667	27930	2306549	1.33
'any'	'any'	'any'	60488	0	8600	9608	2	27931	118147	102.70

6.2 IIPImage With JP2

6.2.1 10 Connections

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
0	10	tile	505	0	41	331	1	5737	39663	0.86
0	10	1024	193	0	250	528	36	6036	117326	0.33
0	10	128	555	0	25	382	5	5570	3313	0.94
0	10	4096	52	0	380	678	41	8301	919556	0.09
10	75	tile	2794	0	39	352	10	7912	42832	4.74
10	75	1024	1103	0	410	652	105	5932	145871	1.87
10	75	128	2848	0	31	362	7	8961	3758	4.83

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
10	75	4096	260	0	1300	1607	406	9392	2144182	0.44
75	-	tile	1294	0	39	278	5	5659	33866	2.20
75	-	1024	531	0	520	762	243	6792	145734	0.90
75	-	128	1330	0	34	302	16	8059	3815	2.26
75	-	4096	137	0	2800	3065	1288	8377	2295748	0.23
'any'	'any'	'any'	11602	0	55	453	1	9392	119108	19.69

6.2.2 100 Connections

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
0	10	tile	2667	0	1700	2763	2	20143	41777	4.53
0	10	1024	1133	0	2100	3212	23	20256	117547	1.92
0	10	128	2667	0	1800	2869	4	20267	3478	4.53
0	10	4096	283	0	2500	3493	42	18249	997249	0.48
10	75	tile	2993	0	1800	2831	8	20211	41844	5.08
10	75	1024	1232	0	1900	3004	130	19020	136719	2.09
10	75	128	3071	0	1900	2924	10	19410	3700	5.21
10	75	4096	327	0	3400	4342	406	15026	1980039	0.56
75	-	tile	1946	0	1700	2781	5	19695	30484	3.30
75	-	1024	885	0	2400	3497	327	20311	151847	1.50
75	-	128	2081	0	1800	2837	16	19714	3839	3.53
75	-	4096	199	0	5000	6176	1855	19963	2151621	0.34
'any'	'any'	'any'	19484	0	1900	2970	2	20311	108729	33.08

6.2.3 1000 Connections

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
0	10	tile	2447	0	29000	29138	30	96348	40797	4.16
0	10	1024	1006	0	29000	29086	298	96476	122311	1.71
0	10	128	2443	0	30000	29535	7	102562	3460	4.15
0	10	4096	249	0	31000	30243	551	99699	958862	0.42
10	75	tile	2610	0	30000	29347	30	101547	41853	4.43
10	75	1024	1019	0	32000	30444	164	98518	138587	1.73
10	75	128	2661	0	30000	29533	17	99714	3791	4.52
10	75	4096	278	0	30000	29741	3147	66154	2136125	0.47
75	-	tile	2610	0	29000	29514	21	102830	31116	4.43
75	-	1024	1076	0	31000	29619	436	99504	146086	1.83
75	-	128	2727	0	31000	29663	20	102842	3786	4.63
75	-	4096	290	0	33000	32406	4012	59935	2182745	0.49
'any'	'any'	'any'	19416	0	30000	29559	7	102842	113628	32.98

6.3 Cantaloupe

6.3.1 10 Connections

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
0	10	tile	2079	126	150	156	14	482	37855	3.53
0	10	1024	902	0	440	440	67	1086	98985	1.53
0	10	128	2234	0	71	79	16	286	3017	3.79
0	10	4096	216	0	2700	2593	501	5047	1274085	0.37
10	75	tile	2480	0	150	153	31	434	37005	4.21
10	75	1024	974	0	570	589	155	1407	108094	1.65
10	75	128	2481	0	89	98	21	389	3219	4.21

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
10	75	4096	232	0	3500	3532	952	6485	1694282	0.39
75	-	tile	1437	0	130	142	28	482	26471	2.44
75	-	1024	613	0	760	795	330	1960	131426	1.04
75	-	128	1493	0	140	151	35	499	3251	2.53
75	-	4096	134	0	4800	4844	1911	8781	1772782	0.23
'any'	'any'	'any'	15275	126	140	330	14	8781	92246	25.92

6.3.2 100 Connections

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
0	10	tile	2224	135	2100	2161	97	6460	37623	3.77
0	10	1024	920	0	4200	4165	570	8275	95181	1.56
0	10	128	2453	0	1300	1434	35	4641	2817	4.16
0	10	4096	233	0	32000	29145	4356	47855	1110176	0.40
10	75	tile	2732	0	2100	2200	160	6028	36088	4.63
10	75	1024	1061	0	5100	5090	1720	9921	110177	1.80
10	75	128	2715	0	1600	1665	86	5195	3140	4.61
10	75	4096	237	0	38000	37170	9407	58394	1719273	0.40
75	-	tile	1309	0	2100	2171	218	4843	27127	2.22
75	-	1024	549	0	6100	6230	2960	11104	127004	0.93
75	-	128	1361	0	2100	2193	229	6166	3236	2.31
75	-	4096	111	2	40000	39677	17361	59647	1645610	0.19
'any'	'any'	'any'	15905	137	2100	3605	35	59647	85546	26.98

6.3.3 1000 Connections

’src min’	’src max’	’derv sz’	’# req’	’# fail’	’median’	’avg’	’min’	’max’	’avg sz’	’req/s’
0	10	tile	1804	240	37000	35236	2214	59561	39718	3.06
0	10	1024	733	65	40000	38536	8021	60352	94972	1.24
0	10	128	1887	164	35000	34279	4191	60246	2858	3.20
0	10	4096	33	182	52000	49261	27089	63969	784414	0.06
10	75	tile	1951	177	37000	36019	6396	60325	39954	3.31
10	75	1024	763	92	41000	39665	8866	61172	112984	1.30
10	75	128	1946	183	35000	34063	3893	58706	3175	3.31
10	75	4096	15	212	52000	49162	15411	61793	1198679	0.03
75	-	tile	1793	166	37000	35616	5143	62728	28640	3.05
75	-	1024	701	94	42000	40405	9708	60055	126595	1.19
75	-	128	1916	162	36000	35219	2672	60437	3296	3.25
75	-	4096	14	163	55000	52021	37471	60095	884227	0.02
’any’	’any’	’any’	13556	1900	37000	35857	2214	63969	38334	23.02

6.4 Loris

6.4.1 10 Connections

’src min’	’src max’	’derv sz’	’# req’	’# fail’	’median’	’avg’	’min’	’max’	’avg sz’	’req/s’
0	10	tile	550	16	97	1849	30	180135	65494	0.95
0	10	1024	202	0	240	1272	5	60368	167608	0.35
0	10	128	517	0	38	2123	4	240225	4359	0.89
0	10	4096	49	0	1300	3730	52	61115	2487976	0.08
10	75	tile	387	0	78	2015	30	120068	59852	0.67
10	75	1024	158	0	220	1863	6	120141	186397	0.27
10	75	128	417	0	26	1201	5	60059	4649	0.72

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
10	75	4096	35	0	1600	1537	25	2740	2706144	0.06
75	-	tile	203	0	73	2593	27	120077	42786	0.35
75	-	1024	98	0	240	346	6	1927	182812	0.17
75	-	128	208	0	20	1926	5	60104	4286	0.36
75	-	4096	20	0	2800	3032	52	6088	2906305	0.03
'any'	'any'	'any'	2844	16	74	1830	4	240225	150819	4.89

6.4.2 100 Connections

'src min'	'src max'	'derv sz'	'# req'	'# fail'	'median'	'avg'	'min'	'max'	'avg sz'	'req/s'
0	10	tile	235	21538	3000	9303	30	123334	58815	1.14
0	10	1024	99	8630	2400	12410	8	123463	147956	0.48
0	10	128	257	21274	2100	8997	5	182996	3972	1.25
0	10	4096	28	2064	3000	13620	56	63701	2104843	0.14
10	75	tile	283	26486	2200	10938	35	182370	52113	1.38
10	75	1024	145	10475	2300	9063	7	68471	176495	0.71
10	75	128	295	26726	1900	9540	5	181271	4328	1.43
10	75	4096	40	2710	4700	17462	126	126689	3337957	0.19
75	-	tile	130	10118	2300	13024	32	123541	39710	0.63
75	-	1024	40	3967	2500	8743	51	123341	181216	0.19
75	-	128	130	9912	1900	11215	6	123242	4481	0.63
75	-	4096	15	1020	3800	8411	960	61979	2583322	0.07
'any'	'any'	'any'	1697	144920	2300	10405	5	182996	185801	8.25