Google Cloud Healthcare API Benchmarking

Experiment and Observation Report

Onix Primary Points of Contact

Scott Cruze
Executive Sponsor
scott.cruze@onixnet.com

Utkarsh Saxena
Project Lead
utkarsh.saxena@onixnet.us

Yasir Drabu, PhD
Principal Advisor
yasir@onixnet.us

Matthew Bolden
Content Strategist
matthew.bolden@onixnet.com

Onix Google Cloud

Table of Contents

Executive Summary

In April of 2020, Google Cloud released its Cloud Healthcare API to the general public as a serverless highly scalable Platform-as-a-Service (PAAS) product to help solve healthcare data interoperability challenges. The Cloud Healthcare API—which is designed to help researchers, developers and data scientists make use of often siloed and disconnected healthcare data—utilizes the most common healthcare data formats, including FHIR, HL7v2 and DICOM. It enables secure and compliant use of healthcare data that is often trapped in disparate health IT systems. Once data is in Google Cloud Platform (GCP), health and life science organizations can seamlessly use that data in analytics (such as ML and AI) and customer applications.

In order to benchmark the Cloud Healthcare API, Google chose Onix (the authors of this paper) to independently design and execute an experiment and document the results. The goal of the experiment was to test the performance and scalability of the Google Cloud Healthcare API at the kind of data volume that a large provider system or health plan might experience. It was decided that **50 million** patients/members was a good representative number.

To execute the experiment, it was necessary to first generate **50 million** synthetic patient records with **26 billion** FHIR resources. We progressively loaded this synthetic patient data — totaling about **60TB** — into a single Google FHIR store observing the response time of eight FHIR APIs (defined in Table 2) as the amount data increased. We chose the eight FHIR APIs as a representative sample of "typical" calls that an application interacting with FHIR data would make.

Throughput

We found that the Cloud Healthcare API and GCP will perform and scale to support very high-volume use cases, as we were able to import **50 million** synthetic patient records (**~26 billion** FHIR resources) into the FHIR store without any optimization for GCP. The FHIR store compressed and stored the raw data effectively, as the final storage size (including space required for indexing) was **~85 TB**. Note that we used the default configuration for GCP, without optimization to more accurately mimic the most common way that GCP users will consume its PaaS capabilities. We were able to import **~1 million** patient records per day in this default configuration, with quota limitations. We estimate that by increasing the import quota the import speed would increase by **100–200%**.

Performance at Scale

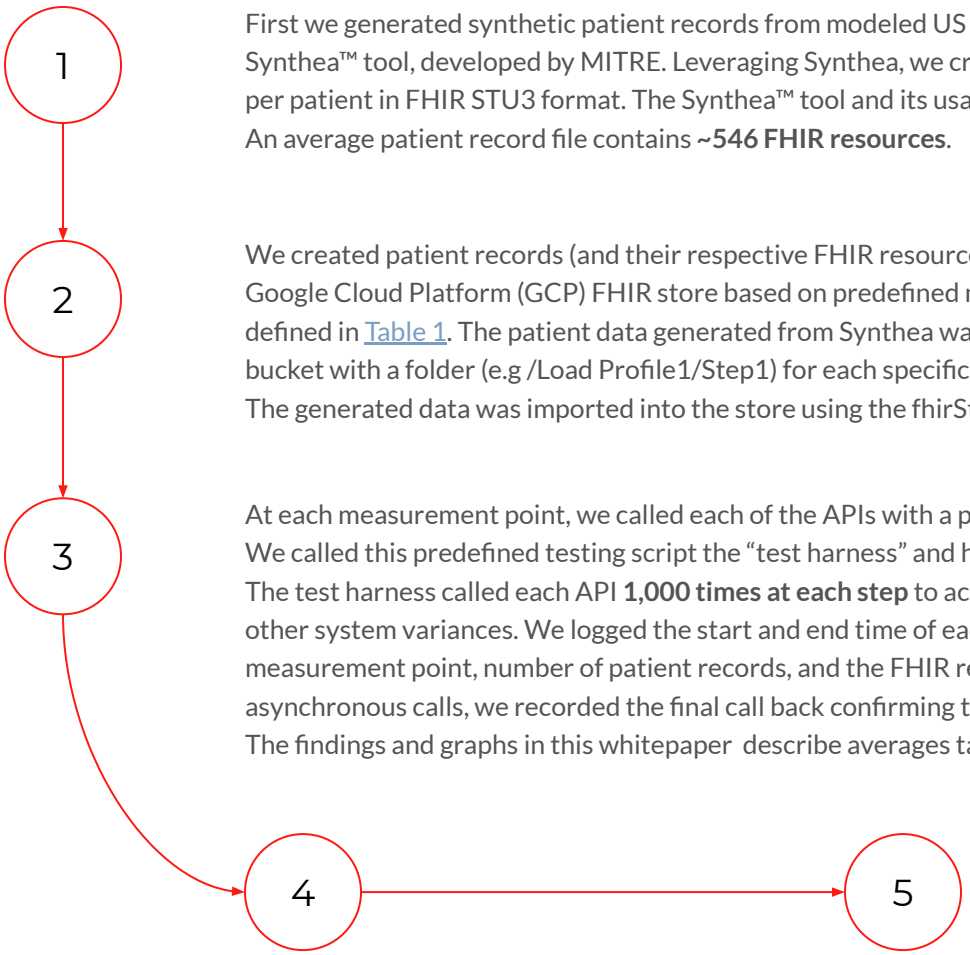
The findings of this experiment show that the Cloud Healthcare API scales in a generally linear way. As the volume of data under management increased, the API's functions remain performant. This conclusion is supported by the performance of the FHIR.Create API (which serves a "write" function); it demonstrated a response time of **< 200 milliseconds** per FHIR resource, even at a volume of **50 million** patients. The FHIR.Read API proved similarly performant, with a response time of **< 140 milliseconds** per FHIR resource at **50 million** patients.

The FHIR.Search API did experience performance degradation at scale (starting at around two million patients, 1B FHIR resources) in our initial, unoptimized test. Noting this degradation, we reran the test at a volume of **50 million** patients against a later version of the Google FHIR API. This resulted in a **381%** speed improvement. It is reasonable to conclude that a similar level of improvement would project backward through the test.

With these findings in mind, we can conclude that the Google Cloud Healthcare API is highly scalable and performant in high-volume use cases.

Experimental Definition

The experimental process that we executed to benchmark performance of the eight selected FHIR APIs involved the following five steps.



1. First we generated synthetic patient records from modeled US census data using the Synthea™ tool, developed by MITRE. Leveraging Synthea, we created one patient record file per patient in FHIR STU3 format. The Synthea™ tool and its usage is detailed in Appendix A. An average patient record file contains **~546 FHIR resources**.

2. We created patient records (and their respective FHIR resources) in batches to load the Google Cloud Platform (GCP) FHIR store based on predefined measurement points as defined in Table 1. The patient data generated from Synthea was stored in a Cloud Storage bucket with a folder (e.g /Load Profile1/Step1) for each specific load point step.
The generated data was imported into the store using the fhirStore.Import API.

3. At each measurement point, we called each of the APIs with a predefined testing script. We called this predefined testing script the "test harness" and have detailed it in Appendix B. The test harness called each API **1,000 times at each step** to account for network latency and other system variances. We logged the start and end time of each API call with the measurement point, number of patient records, and the FHIR resource count. In the case of asynchronous calls, we recorded the final call back confirming the operation has completed. The findings and graphs in this whitepaper describe averages taken from these 1,000 calls.

4. The final reading for each step, which was an average of the 1,000 calls at each step, was recorded in BigQuery for visualization and analysis.

5. The response time was recorded on a per patient record and per FHIR resource (within a patient record) level.

We divided the measurement points into four load profiles: LP1 to LP4. Each load profile had the same step size within that particular group (for example, 10K in LP1). Each ensuing group had a larger step size (in terms of patient records) than the previous group.

Load Profile	Load Size	Step Size	Description
LP1	10K – 100K	10K patients	Add patients in step sizes of 10,000 and measure at each step
LP2	100K – 1M	50K patients	Add patients in step sizes of 50,000 and measure at each step
LP3	1M – 10M	500K patients	Add patients in step sizes of 500,000 and measure at each step
LP4	10M – 50M	2M patients	Add patients in step sizes of 2M and measure at each step

Table 1: Measurement Points

The Cloud Healthcare API REST interface consists of many different methods so it was impractical to test them all. We selected eight API methods that are representative of the most commonly used interactions required to support the loading and consumption of FHIR data.

Load Profile Load Size	Step Size Description
fhir.create	Creates a FHIR resource within a FHIR store
fhir.read	Retrieves a FHIR resource from a FHIR store
fhir.conditionalDelete	Deletes FHIR resources that match a search query against a FHIR store
fhir.search	Retrieves FHIR resources from a FHIR store that match a search query.
fhir.conditionalUpdate	Performs a search query against a FHIR store and updates the returned FHIR resource (if found) or creates a new FHIR resource (if the search query returns no results)
fhir.conditionalPatch	Performs a search query against a FHIR store and updates parts of the returned FHIR resource (if found)
fhirStores.import	Imports FHIR resources to a FHIR store by loading data from the specified sources
fhir.executeBundle	Executes a (transaction) bundle of FHIR resources, each of which represents an operation, such as create, update, or delete, on a resource in a FHIR store

Table 2: Google Cloud Healthcare API: FHIR APIs Used in the Experiment

Experiment Execution

Prerequisites

At a high level, we needed the following to run the experiment:

1. Google Compute Engine to run the Synthea tool to generate the synthetic patient data.
2. Google Cloud Storage to store the output of the Synthea files in a persistent manner.
3. Compute instance to run the load testing scripted test cases (using Python).
4. Google BigQuery to store the results of the test.
5. A data visualization tool to summarize, visualize and host the dashboard.
6. GCP project to store the generated raw data.
7. GCP project with Cloud Healthcare API enabled to store the FHIR data.

Google Cloud Platform (GCP) Accounts

To run the experiment, we set up two GCP accounts:

The first account was provided by Onix for the generation of the synthetic data into a GCP bucket. In this account, we set up:

A. VM/Compute n1-standard-16 (16 vCPUs, 60 GB memory), which was used to install and run Synthea and the test harness.

B. Google Cloud Storage to store the synthetic patient data, which was estimated to be 60TB.

C. Big Query to store the response times and related observations

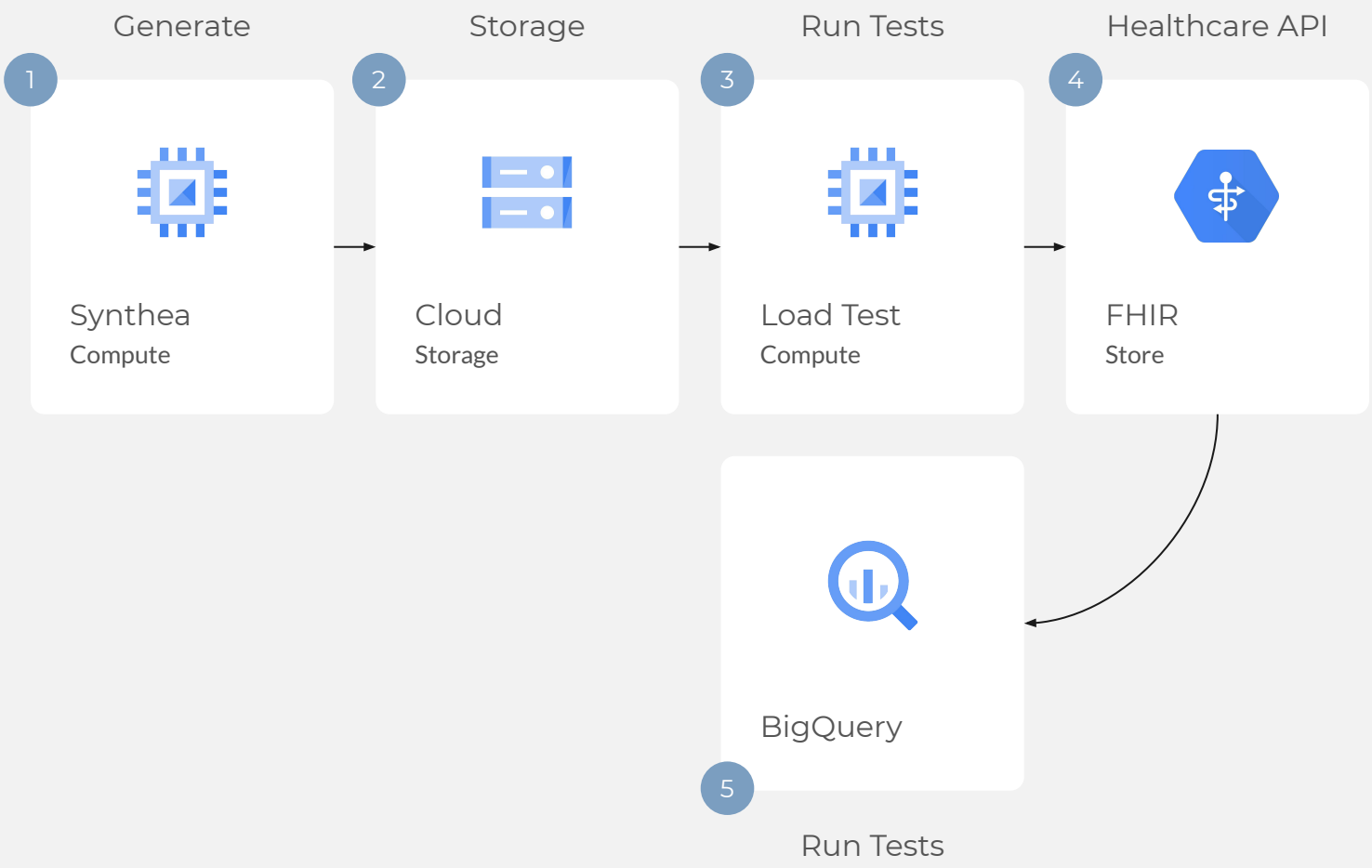
The second account was provided by Google Cloud, wherein we:

A. We set up the FHIR store in this account.

B. This store was not optimized for any specific indexes. We used default settings.

Execution

We set up the experiment as shown in Figure 1 below.
There were **5 key execution steps** as documented in Table 3, also below.



Google Cloud Platform

Figure 1: Experiment execution architecture

Step	Action	Description
1	**Launch Compute Instances**	Install Synthea on multiple compute instances to generate data in parallel.
2	**Create Storage**	Store the synthetic patent data files.
3	**Run Synthea**	Conduct multiple runs of the n1 instance to generate data for multiple states and store it in separate folders in the connected Cloud Storage bucket. We mapped the Cloud Store bucket as a volume on the compute instance and generated data in load profile volumes and steps
4	**Data Import & Execute Test Scripts**	Import patient data set of incremental batches (as mentioned in Measurement Points) to FHIR stores and measure the performance and save the results in BigQuery with respect to the import function. This was done across different GCP projects, so setting the right permissions was important. Run queries as mentioned in FHIR APIs Under Consideration (Table 2).
5	**Measure Performance**	Collect data size and response time for each endpoint under consideration into BigQuery.

Table 3: Experiment Execution Key Steps

The overall experiment metrics are described below:

1. We ran the experiment through **50M** patient records (or **~26 billion** FHIR resources) for all the APIs defined in Table 2.

2. Overall, we ran the test harness **57,000** times. Each API was called **1,000** times at each measurement point for a total of **57** points. The **57** points in the ensuing graphs represent the **average** of the 1,000 calls.

3. We collected **57** average response time readings for each API for Load profiles 1, 2, 3, and 4.

   For Load Profile-1 (10k-100k) we ran 10 experiments with step size of **10K** patient records.

   For Load Profile-2 (100k-1M) we ran 18 experiments with step size of **50k** patient records.

   For Load Profile-3 (1M-10M) we ran 9 experiments with step size of **1M** patient records.

   For Load Profile-4 (10M-50M) we ran 20 experiments with step size of **2M** patient records.

4. We then calculated the maximum, median, minimum and 95th percentile response times for each API.

Experiment Results: Overview

Summary

The measurements of the average response time for each API in terms of FHIR resources are tabulated on page 10. The responses at each measurement point are detailed in the following sections, which are grouped by the specific API.

Table 6 shows the throughput performance of the Import function. Table 7 shows the performance of the Execute Bundle as compared to the Import function. These are the two primary API functions to load the FHIR store.

Key Numbers of Interest:

1

The total size of the generated source data (JSON files on Google Cloud Storage) for **50 million** patients was **60TB** and was comprised of **~26 billion** FHIR resources.

2

The Import operation to load the generated source data from Google Cloud Storage into the Cloud Healthcare FHIR API took about **50 days** to complete.

We imported **50 million** patient records, or about **~26 billion** FHIR resources.

3

The Cloud Healthcare FHIR API compressed and stored the source data effectively requiring **~85TB** of storage, including space required for indexing.

4

The response time for all API methods remained fairly linear, with the exception of the Search API, and well within a reasonable range.

NOTE —After completion of this testing Google released a performance improvement to the Search API. We re-ran the now-optimized Search API with the performance improvement on the datapoint "Load Profile 4 / 50 million patient records" and included the result in Table 5 below.

As the data outlined and depicted below will show, **the Healthcare API and the GCP FHIR store maintained performance and scaled well across all APIs and load profiles**. The first depiction of the results are provided in tabular format on page 10. Graphical representations of the results are provided on pages 11-19.

Results: API Latency

API Name	Scope	Min	Median	Max	95th
Create	Avg. seconds per FHIR resource	0.091	0.117	0.272	0.257
Read	Avg. seconds per FHIR resource	0.074	0.096	0.153	0.147
Delete	Avg. seconds per FHIR resource	0.089	0.143	0.229	0.220
Conditional Update	Avg. seconds per FHIR resource	0.127	0.320	0.885	0.829
Conditional Patch	Avg. seconds per FHIR resource	0.128	0.233	0.458	0.436
Conditional Delete	Avg. seconds per FHIR resource	0.118	0.439	0.932	0.883
Search	Total search duration in seconds	0.437	2.755	10.18	9.440

Table 4: Average FHIR resource time in seconds for each API - Minimum, Median, Maximum and 95th percentile

Results: Search Retest with Latest Release of API

API Name	Scope	Before: Initial Release Run (seconds)	After: Latest Release Run (seconds)	Speed Improvement
Search (Rerun)	Average search duration in seconds	4.066	1.066	381%

Table 5: Total search duration in seconds. Search was rerun with performance optimization on the full 50 million patient record dataset. This table shows the pre- and post-optimized figures for comparison.

Results: Import Throughput

API Name	Scope	Min	Median	Max	95th
Import	FHIR resources per second	1617	5461	10547	10038

Table 6: Number of FHIR resources retrieved/reviewed per second for each API - Minimum, Median, Maximum and 95th percentile

Results Comparing Import Vs ExecuteBundle

API Name	Scope	Min	Median	Max	95th
Execute Bundle	Avg milliseconds per FHIR resource	1.481	5.536	22.267	18.258
Import	Avg milliseconds per FHIR resource	0.095	0.183	0.722	0.668

Table 7: Time taken to import each FHIR resource using the Execute Bundle and Import APIs. Importing is significantly faster.

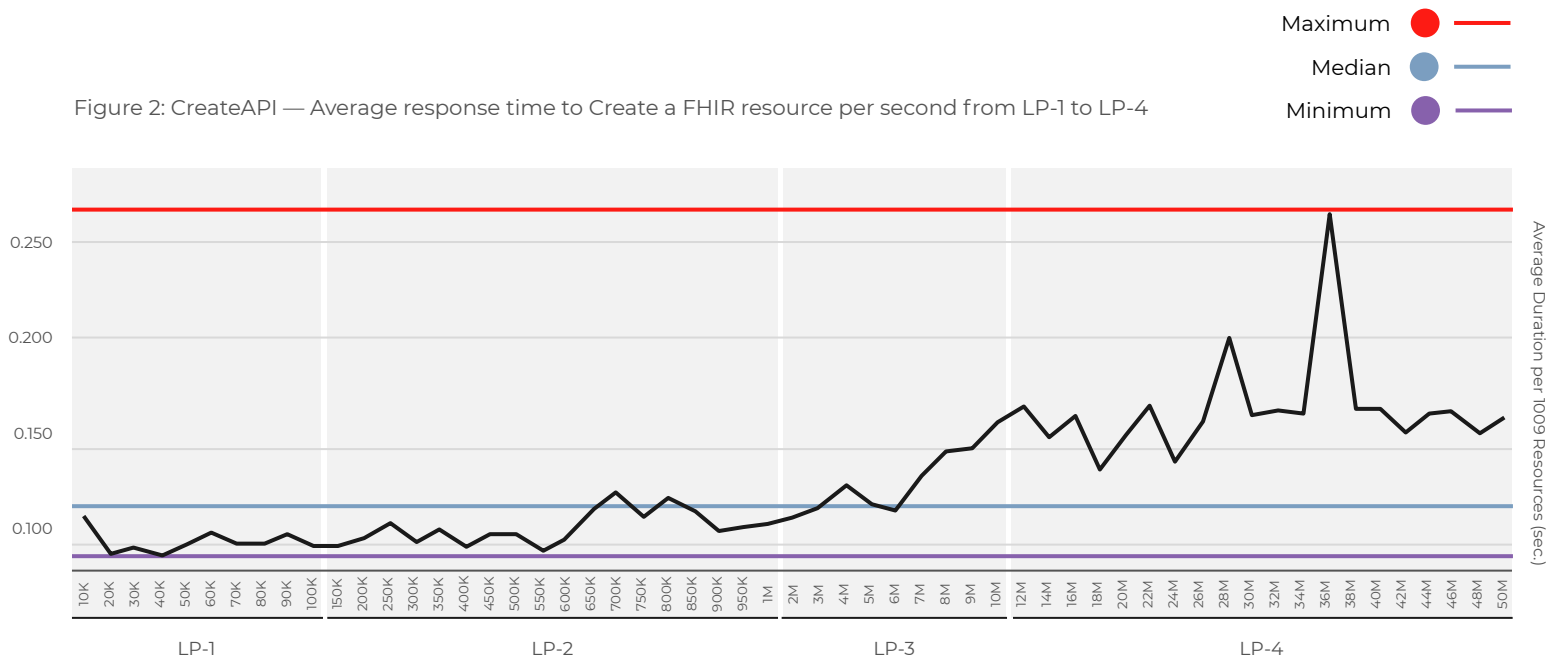
Experiment Results: Individual APIs

FHIR.Create (link to documentation)

For each measurement point, we executed the Create test harness script that created **~1,000 FHIR resources** in the FHIR store. Each data point is an average of the **1,000 Create response times**. After the resources were created and response times measured, we reset the store as the test harness deleted the created resources.

The FHIR.Create API demonstrated impressive performance at scale. At a volume of 50 million patients, the API showed a response time of significantly less than 200 milliseconds per FHIR resource. The spike in response time at 36 million patients appears to be an anomaly, as overall there is little variation.

Figure 2: CreateAPI — Average response time to Create a FHIR resource per second from LP-1 to LP-4

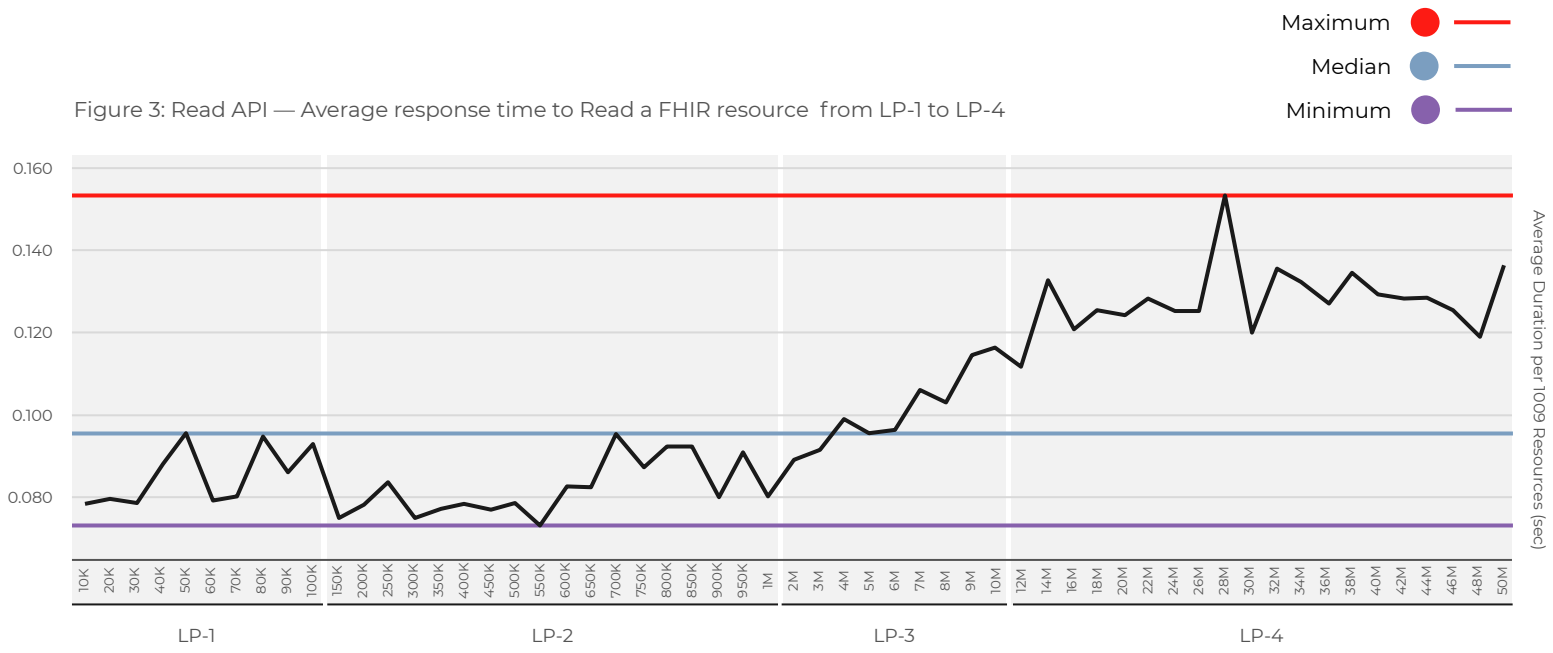


FHIR.Read (link to documentation)

For each measurement point, we executed the Read test harness script that read **~1,000** FHIR resources in the FHIR store. Each data point is an average of the **1,000** Read response times.

As evidenced by the graph below, the API is very responsive at scale. At 50 million patients, FHIR.Read demonstrates a response time of < 140 milliseconds per FHIR resource.

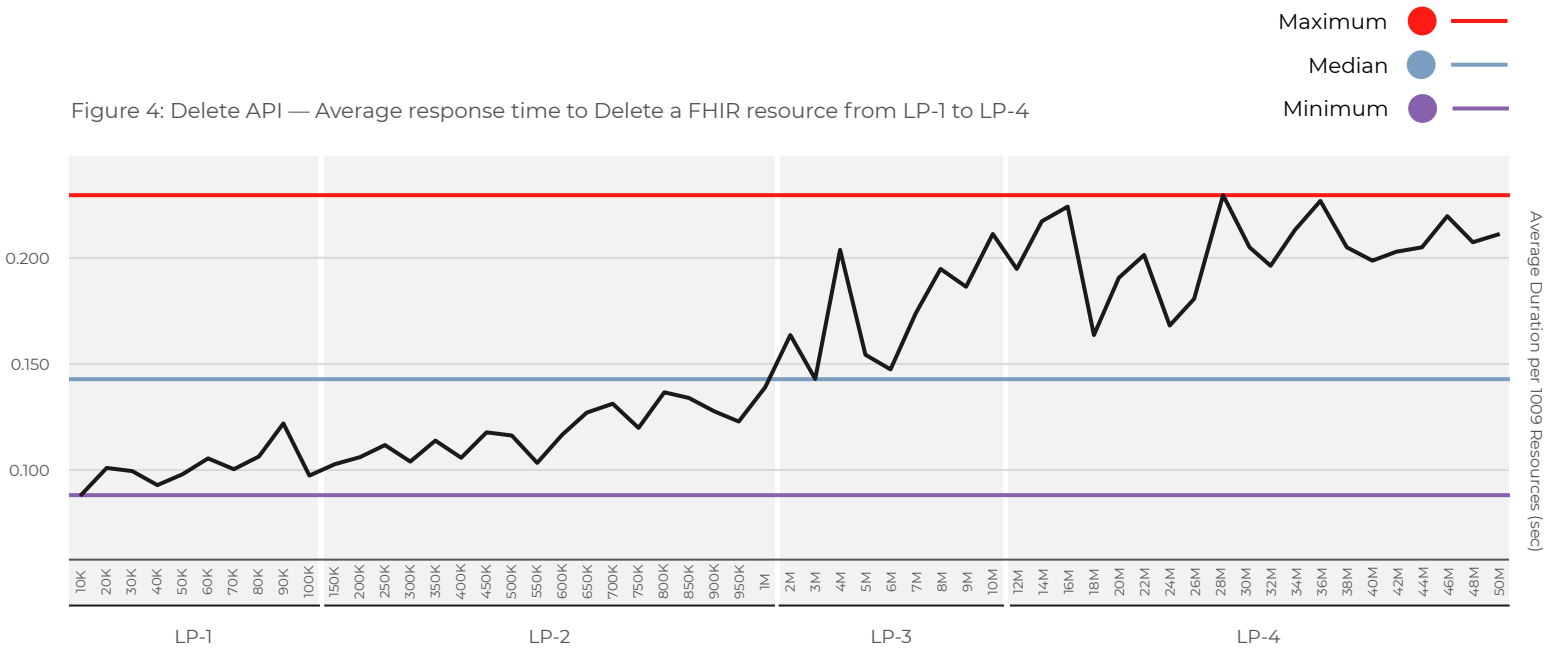
Figure 3: Read API — Average response time to Read a FHIR resource from LP-1 to LP-4



FHIR.Delete (link to documentation)

For each measurement point, we executed the Delete test harness script that deleted **~1,000** FHIR resources in the FHIR store. Each data point is an average of the **1,000** Delete response times. After the measurements were taken, the FHIR store was reset by loading those resources back into it.

Figure 4: Delete API — Average response time to Delete a FHIR resource from LP-1 to LP-4



FHIR.Search: Initial Release(link to documentation)

For each measurement point, we executed the Search test harness script that performed the following actions on the FHIR store:

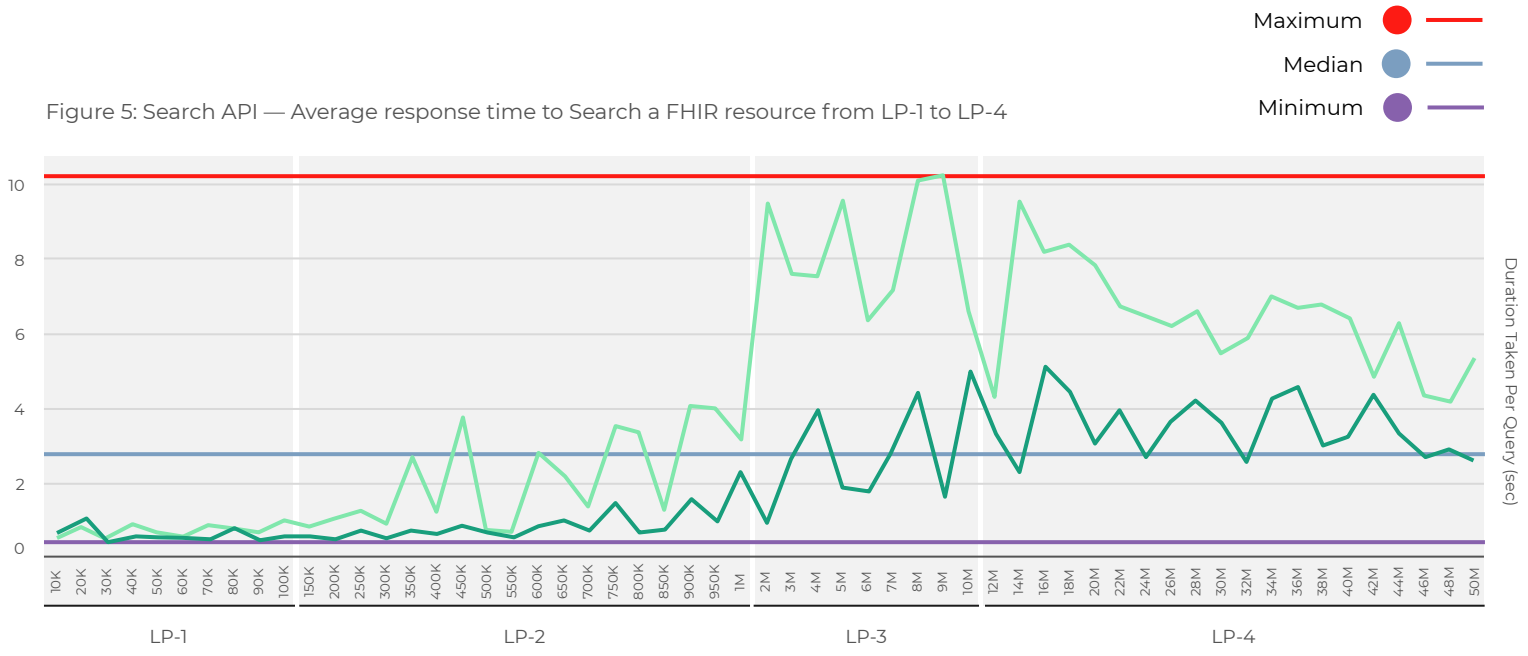
1. Executed queries using the Concepts and Attributes fields created by the Synthea synthetic data generator tool.
2. Searched FHIR resources in the data store using search modifiers such as: missing, :exact, :contains, :text, :in, :not in , :above, :below, :[type], :not, and :recurse.

Example Query Parameters:

- Patient is female from Alabama
- Patient is male

Noting performance issues beginning in LP-3 in our initial test, we re-ran the test with a later release of the Google FHIR API at a volume of 50 million patients. This second test demonstrated a 381% improvement in response time, as shown in the table below the graph. We believe that a similar level of improvement would project backward through the entire test.

Figure 5: Search API — Average response time to Search a FHIR resource from LP-1 to LP-4



FHIR.Search: against latest release of Google FHIR API

API Name	Scope	Before: Initial Version Run (seconds)	After: Latest Version Run (seconds)	Speed Improvement
Search (Rerun)	Average search duration in seconds	4.066	1.066	381%

Total search duration in seconds. Search was rerun against the latest version of the Google FHIR API on the full 50 million patient record dataset. This table shows the pre- and post- figures for comparison.

FHIR.ExecuteBundle (link to documentation)

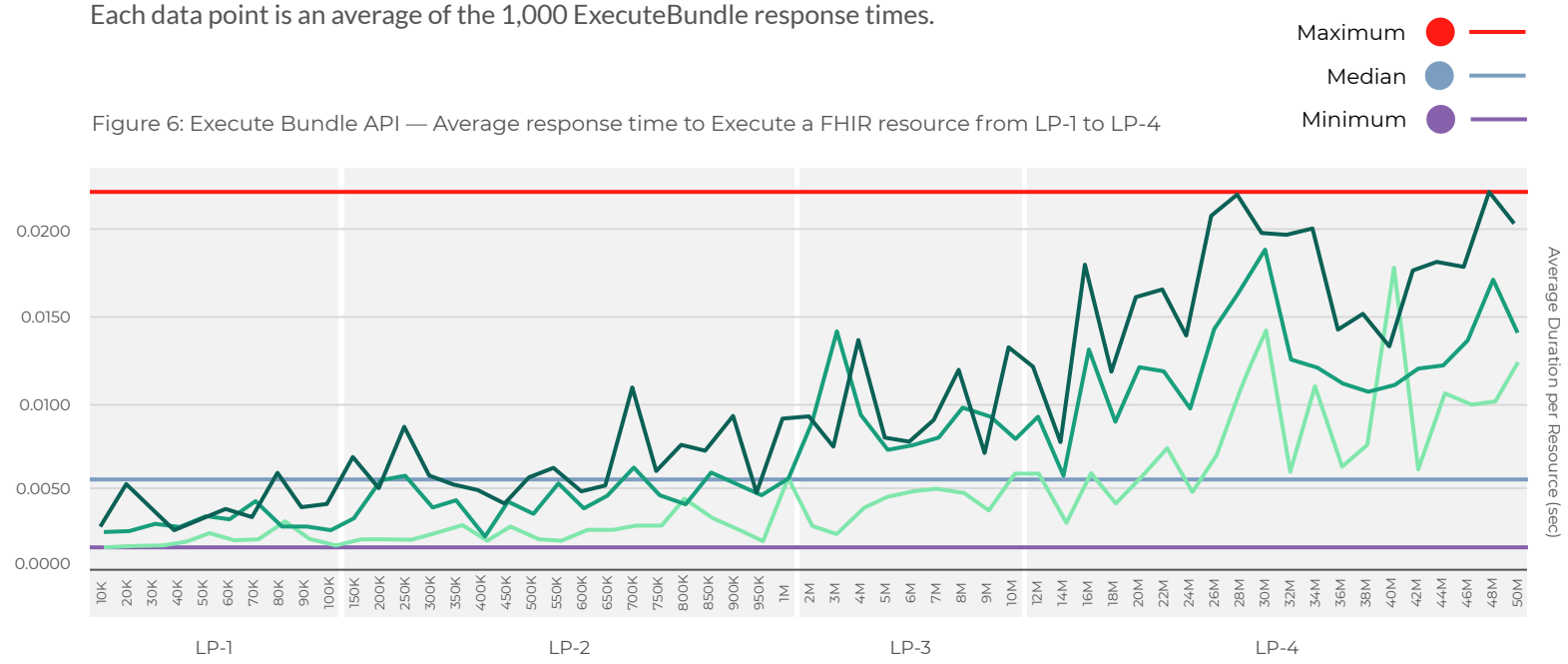
For each measurement point, we executed a test script that created FHIR resources using three distinct patient bundle sizes (based on number of FHIR resources per bundle) on the FHIR store. They were:

- Small Bundle — 312
- Medium Bundle — 546
- Large Bundle — 1789

We then completed the transaction interaction on the FHIR Store.

Each data point is an average of the 1,000 ExecuteBundle response times.

Figure 6: Execute Bundle API — Average response time to Execute a FHIR resource from LP-1 to LP-4



FHIR.ConditionalUpdate (link to documentation)

For each measurement point, we executed the test harness script that performed the following action on the FHIR store:

1. We conditionally updated 1,000 FHIR resources at each measurement point.
2. We recorded actual and average time to Update the FHIR resource.

There are two conditions:

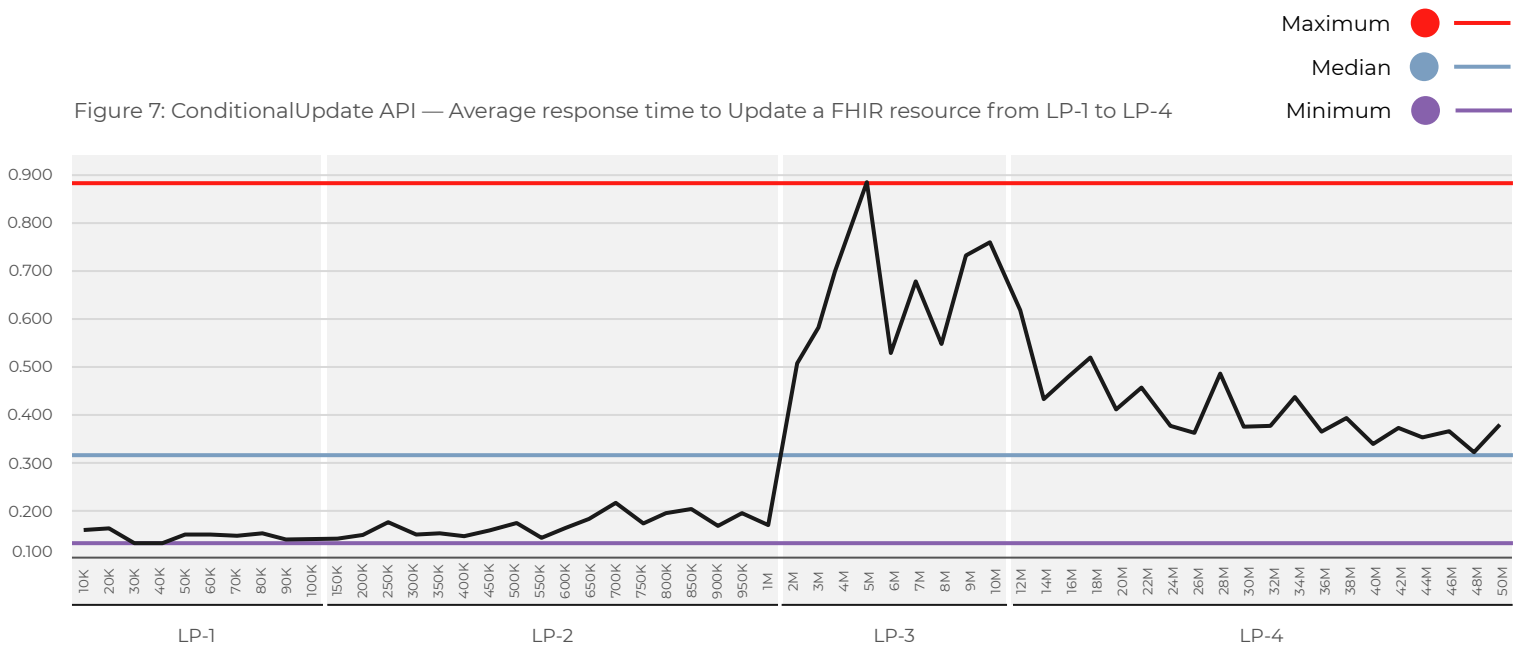
A. Patient gender has been updated

B. Practitioner gender has been updated

Each data point is an average of the 1,000 ConditionalUpdate response times.

Note: The results displayed here were from the initial test of the ConditionalUpdate API. We saw a performance degradation at the same data volume as we saw with the Search API. Given that conditional operations are effectively a search followed by an operation, it is reasonable to conclude that ConditionalUpdate performance would improve significantly when using the latest release of the Cloud Healthcare API with improved Search performance.

Figure 7: ConditionalUpdate API — Average response time to Update a FHIR resource from LP-1 to LP-4



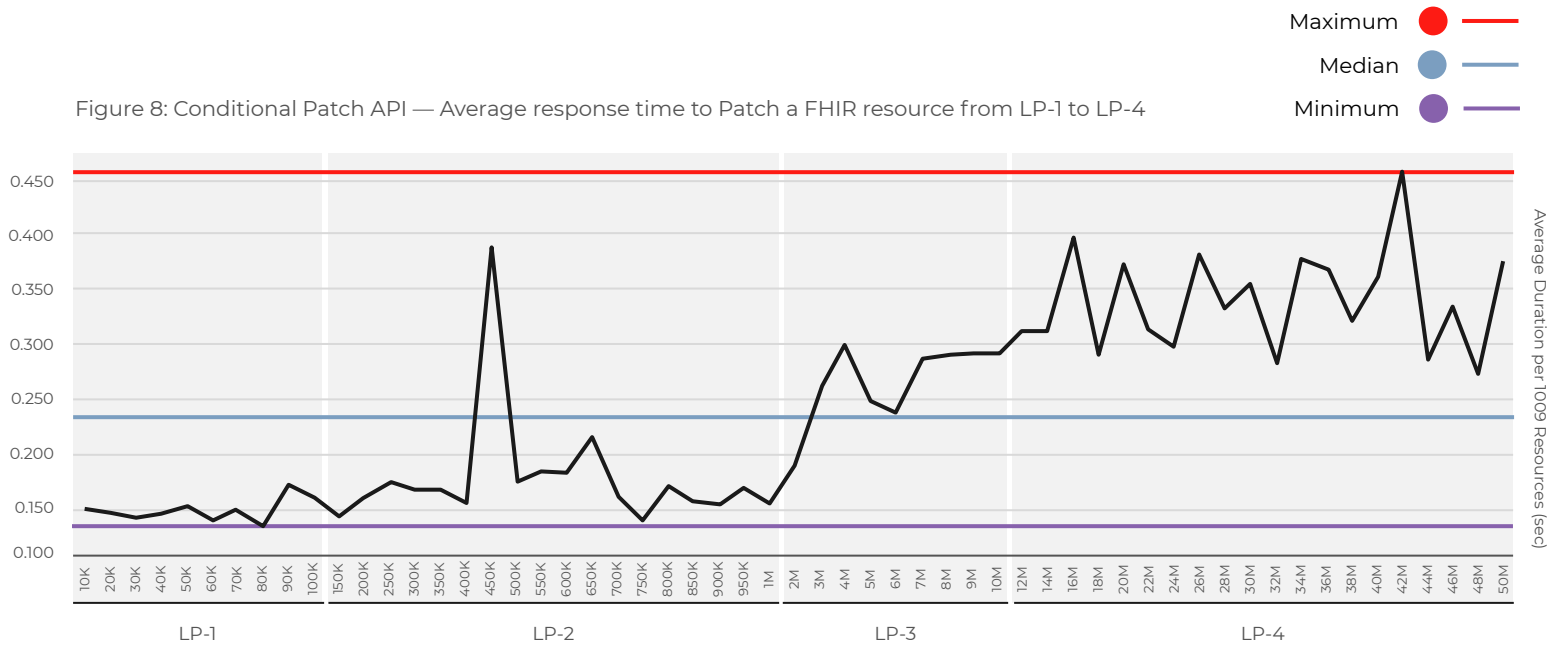
FHIR.ConditionalPatch (link to documentation)

For each measurement point, we executed the test harness script that performed the following action on the FHIR store:

1. Conditionally patched 1,000 FHIR resources at each measurement point by doing random modifications.
2. Recorded actual and average time to Patch the FHIR resource.

Each data point is an average of the 1,000 ConditionalPatch response times.

Figure 8: Conditional Patch API — Average response time to Patch a FHIR resource from LP-1 to LP-4



FHIR.ConditionalDelete (link to documentation)

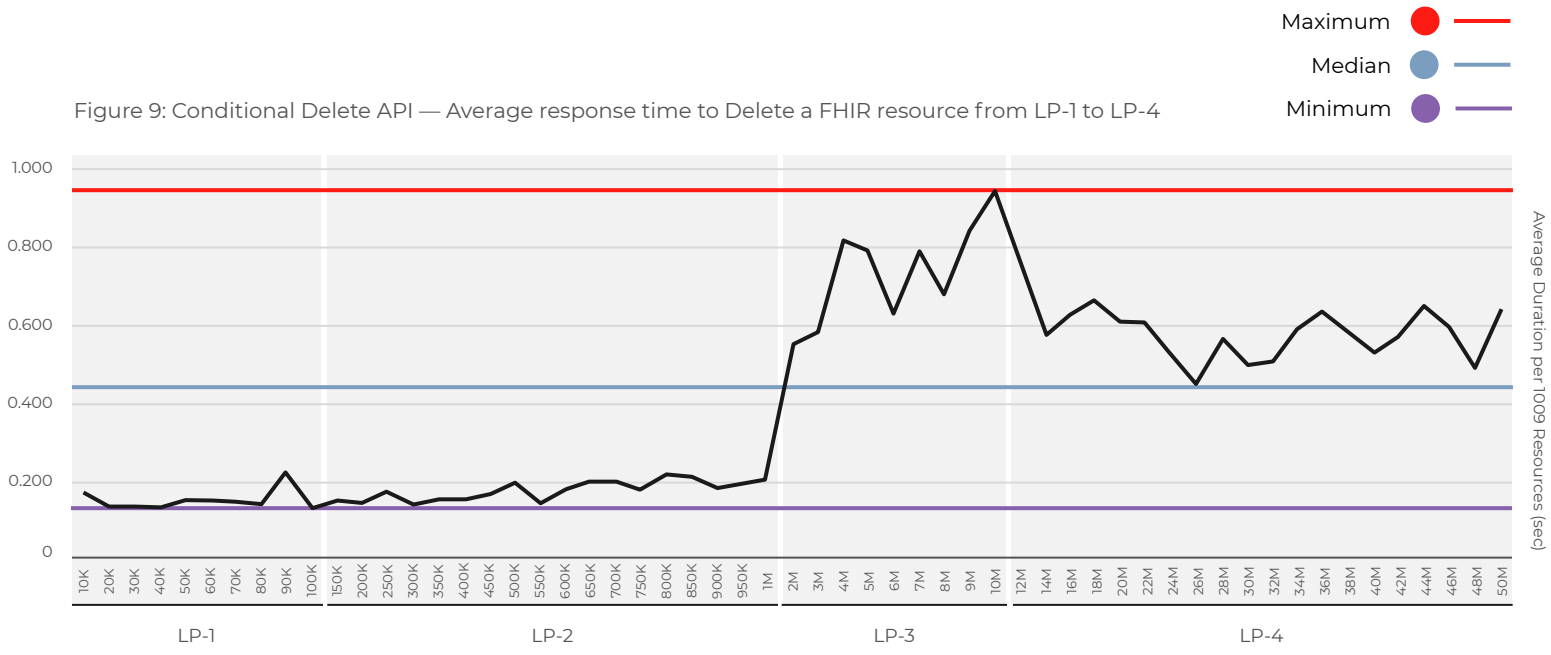
For each measurement point, we executed the test harness script that performed the following action on the FHIR store:

1. Conditionally Deleted 1,000 FHIR resources at each measurement point by doing random deletions.
2. Recorded actual and average time to Delete the FHIR resource.

Each data point is an average of the 1,000 Delete response times.

Note: The results displayed here were from the initial test of the ConditionalDelete API. We saw a performance degradation at the same data volume as we saw with the Search API. Given that conditional operations are effectively a search followed by an operation, it is reasonable to conclude that ConditionalDelete performance would improve significantly when using the latest release of the Cloud Healthcare API with improved Search performance.

Figure 9: Conditional Delete API — Average response time to Delete a FHIR resource from LP-1 to LP-4



FHIR.Import (link to documentation)

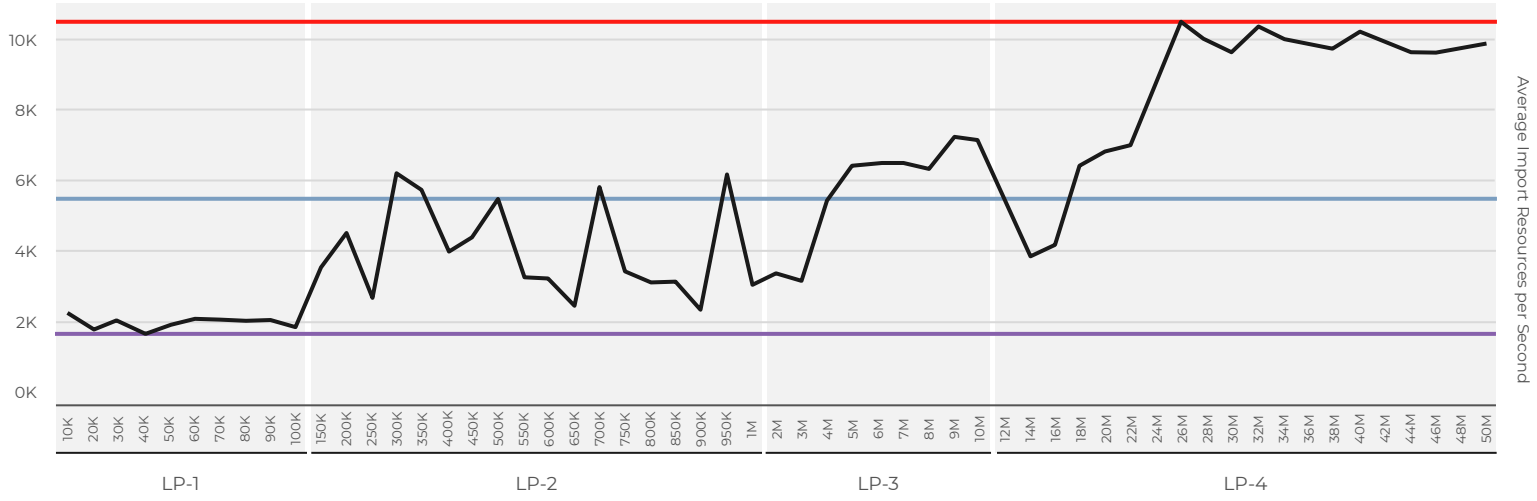
For each measurement point, we executed the test harness script that performed following action on FHIR store:

1. Imported patient records at each measurement point.
2. Recorded actual and average time to Import the patient record

Each data point is indicated in terms of resources per second.

Note: Performance of the FHIR.Import API was limited by quota caps until the 26 million patient load. Prior to the 26 million patient load point, Import was capped at 0.5 GB/minute. At 26 million patients, quota was increased to 1.5 GB/minute. This explains the significant increase of resources per second that occurred at that point.

Figure 10: Import API — Average FHIR resources Imported per second LP-1 to LP-4



Conclusion

Upon analysis of our results, we found that the Cloud Healthcare API and GCP FHIR API scale and maintain performance up to 50 million patient records (or **~26 billion** FHIR resources and comprising **85 TB** of data) in the FHIR store.

We were able to import **~1 million** patients per day (and 10,000 FHIR resources per second) into the FHIR store, reaching the **50 millionth** patient record in **50.23 days**. We accomplished this using the default (unoptimized) GCP configuration and sequential loading. We are confident that with optimization and parallelization, the import speed would be increased by at least 2-3x.

Importantly, the Cloud Healthcare API and GCP FHIR API scale in a generally linear way, and remain performant in high-volume use cases. This is evidenced by the fact that performance remains consistent as the data volume scales to 50 million synthetic patients. For example, the FHIR.Create API demonstrated a response time of < 200 milliseconds per FHIR resource at a volume of 50 million patients, while the FHIR.Read API exhibited a response time of < 140 milliseconds per FHIR resource at the same load volume.

In addition to the fast import speed and performance-at-scale of the Cloud Healthcare API and the GCP FHIR store, we would also like to point out that the Cloud Healthcare API provides a fully managed development environment; configuring the GCP test environment only took a few minutes. We were also able to spin up and tear down resources in a matter of seconds.

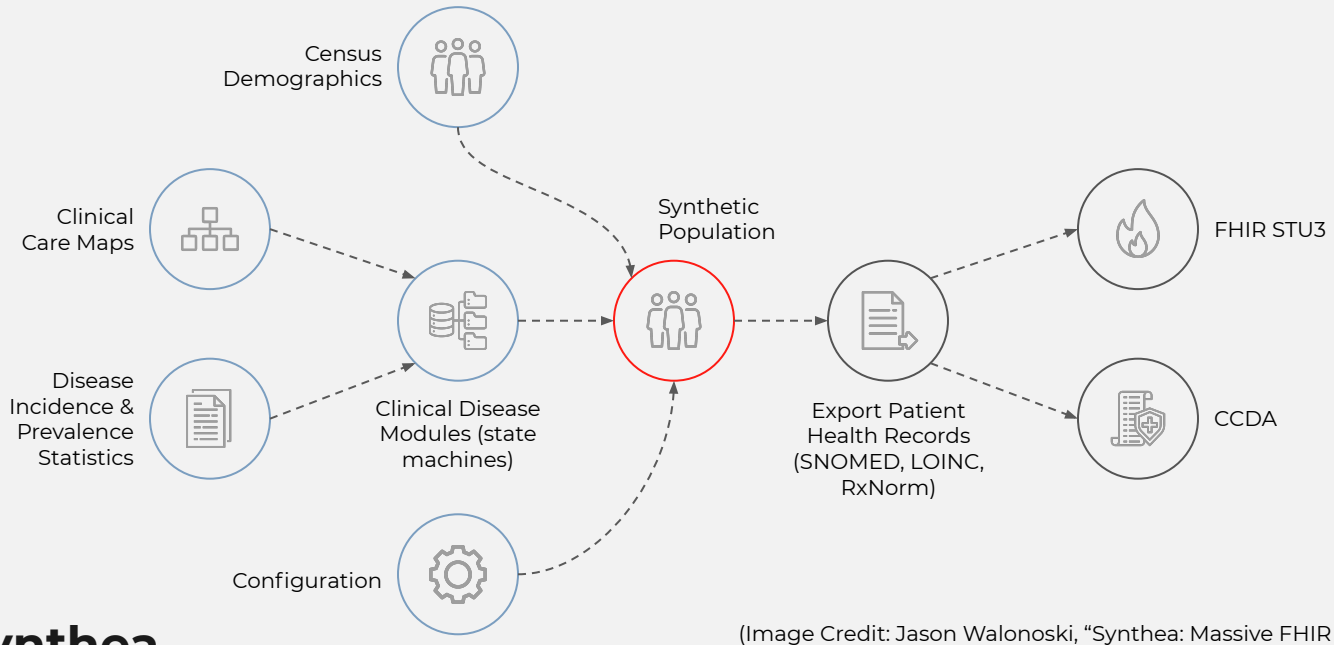
Even though HIPAA compliance was not a factor due to our using synthetic patient data, security and compliance is built into GCP, and all data in Google Cloud is encrypted in transit and at rest. Leveraging this fully managed environment helped to save time and resources, allowing our team to focus on the experiment rather than acquiring hardware and hosting and maintaining the environment.

We conclude that the Cloud Healthcare API is scalable, fast, and quick to configure, enabling researchers, developers, data scientists to rapidly build intelligent healthcare solutions in the cloud.

Appendix A — Synthea

Overview

Synthea™ is an open-source Synthetic Patient Population Simulation developed by The MITRE Corporation that is used to generate the synthetic patients. It generates realistic-looking (but not real) patient data modeled on US Census data.



Synthea

(Image Credit: Jason Walonoski, "Synthea: Massive FHIR Data" presentation at HL7 FHIR DevDays 2018)

Currently, Synthea features:

- Birth to Death Lifecycle
- Configuration-based statistics and demographics (defaults with nationwide US Census data)
- Drop-in Generic Modules
- Primary Care Encounters, Emergency Room Encounters, and Symptom-Driven Encounters
- Formats - HL7 FHIR and HL7 C-CDA
- Conditions, Allergies, Medications, Vaccinations, Observations/Vitals, Labs, Procedures, CarePlans

Generating Synthetic Patient Datasets

We used the following operations to generate synthetic patient datasets with Synthea. Note that Synthea requires Java 1.8 or above.

Steps:

1. Clone the Synthea repo, then build and run the test suite:

```
git clone https://github.com/synthetichealth/synthea.git
cd synthea
./gradlew build check test
```

2. Generate population for each state:

```
./run_synthea
run_synthea [-s seed] [-p populationSize] [-m moduleFilter] [state [city]]
```

For e.g the command to generate 500,000 patient records having gender as "Male" from "Dallas" City of "Texas" State:-

```
./run_synthea
run_synthea -s 2345627 -p 500000 -g M Texas "Dallas"
```

Synthetic Patient Data Sets

We generated **50 million** patient records for this experiment in FHIR STU3 format. In order to generate a patient population resembling the demographic and health diversity of the US population, we generated the 50 million patient records using **16%** of the Census population of each state. We did this to ensure that the number of FHIR resources contained within the 50 million patient records would resemble the US population and its associated amount of data. See "Data Usage Statistics on GCP" section below for more information.

- State or Jurisdiction — The US state or jurisdiction simulated
- Estimated Population — The number of patient records generated per simulation which has been gradually increased per state
- Generated Records — Total number of patient records generated per state

State or Jurisdiction	Est. Pop 2019	Generated Records
California	39,512,223	6,321,956
Texas	28,995,881	4,639,341
Florida	21,477,737	3,436,438
New York	19,453,561	3,112,570
Pennsylvania	12,801,989	2,048,318
Illinois	12,671,821	2,027,491
Ohio	11,689,100	1,870,256
Georgia	10,617,423	1,698,788
North Carolina	10,488,084	1,678,093
Michigan	9,986,857	1,597,897
New Jersey	8,882,190	1,421,150
Virginia	8,535,519	1,365,683
Washington	7,614,893	1,218,383
Arizona	7,278,717	1,164,595
Massachusetts	6,892,503	1,102,800
Tennessee	6,829,174	1,092,668
Indiana	6,732,219	1,077,155
Missouri	6,137,428	981,988
Maryland	6,045,680	967,309
Wisconsin	5,822,434	931,589
Colorado	5,758,736	921,398
Minnesota	5,639,632	902,341
South Carolina	5,148,714	823,794
Alabama	4,903,185	784,510
Louisiana	4,648,794	743,807
Kentucky	4,467,673	714,828
Oregon	4,217,737	674,838
Oklahoma	3,956,971	633,115
Connecticut	3,565,287	570,446
Utah	3,205,958	512,953
Iowa	3,155,070	504,811
Nevada	3,080,156	492,825
Arkansas	3,017,804	482,849
Mississippi	2,976,149	476,184
Kansas	2,913,314	466,130
New Mexico	2,096,829	335,493
Nebraska	1,934,408	309,505
West Virginia	1,792,147	286,744
Idaho	1,787,065	285,930
Hawaii	1,415,872	226,540
New Hampshire	1,359,711	217,554
Maine	1,344,212	215,074
Montana	1,068,778	171,004
Rhode Island	1,059,361	169,498
Delaware	973,764	155,802
South Dakota	884,659	141,545
North Dakota	762,062	121,930
Alaska	731,545	117,047
Vermont	623,989	99,838
Wyoming	578,759	92,601
Total	**327,533,774**	**52,405,404**

Appendix B — Test Harness

The test harness executes tests by using a python test library and then generates reports.
The test harness contains all the information needed to compile and run a test, including FHIR store details, Bundle size, Queries etc.

We created the test harness and ran the experiments for the below APIs:

Create FHIR

```
request = service.projects().locations().datasets().FHIRStores().FHIR().create(parent=parent,
type=type_, body=http_body_body)
response = request.execute()
```

Delete FHIR

```
request = service.projects().locations().datasets().FHIRStores().FHIR().delete(name=name)
response = request.execute()
```

Read FHIR

```
request = service.projects().locations().datasets().FHIRStores().FHIR().read(name=name)
response = request.execute()
```

Search FHIR

```
request = service.projects().locations().datasets().FHIRStores().FHIR().search(parent=parent,
body=search_resources_request_body)
response = request.execute()
```

Execute Bundle

```
request = service.projects().locations().datasets().FHIRStores().FHIR().executeBundle(parent=parent,
body=http_body_body)
response = request.execute()
```

Appendix C — Experimental Considerations

Results were generated with verbose mode – OFF. With verbose mode – ON, the names of users are printed in the terminal while generating.

We used Google Cloud Platform (GCP) in its default setting (without any optimization) to generate the data. Our GCP configuration was as follows:

- CPU - 16 Core
- RAM - 60 GB
- Hard Disk - 1 TB SSD

Since the Synthea source code runs on JVM, memory allocation is done automatically. It is recommended to have multiple cores to generate the data.

CPU Intensive Understanding

- It is also recommended that we use multiple CPU-intensive machines to share the load while generating the synthetic patient data.
- To distribute the data geographically it is recommended to have multiple non-CPU intensive machines to test the GCP FHIR API.

“Pretty-Print” vs “non-pretty” JSON bundles

- “Non-pretty” JSON bundles are faster to import, but Synthea only generates “Pretty-Print” JSON bundles. We observed that the time-savings benefit of importing “non-pretty” JSON bundles would be outweighed by the extra time it took to convert “Pretty-Print” JSON bundles to “non-pretty” JSON bundles for import.

Tip: remember to check your FHIR store to ensure that all FHIR resources are created in each patient record.