



Google
Developers



Optimizing Your App Engine App

Marzia Niccolai
Spender of GBucks

Greg Darke
Byte Herder

Troy Trimble
Professional Expert

Agenda

- Overview
- Writing applications efficiently
 - Datastore Tips
 - Caching, Caching, Caching
 - Batch API requests
 - Asynchronous workflows
 - Offline workflows
- Performance Settings
- Future Work
 - App Engine Servers





Overview

Latency vs. Cost

Overview

- Latency vs Cost is a central tradeoff
- More efficient = lower costs
- More resources = higher costs
- Sometimes, you can reduce costs at the expense of speed



What is "Good Latency"

Overview

Current Load ?				
URI	Req/Sec current	Requests last 17 hrs	Runtime MCycles last hr	Avg Latency last hr
/api/int/gap2/artwork/	1.0	86.72K	196	232 ms
/current_user/	0.7	54.47K	612	420 ms
/pano/	0.3	19.05K	249	172 ms
/api/int/gap2/canonical_artist/	0.2	12.67K	35	75 ms
/	0.2	7.46K	13	46 ms
/api/int/gap2/asset/	0.1	4.71K	407	270 ms
/api/int/gap2/user_bookmarkgroup/	0.1	3.19K	144	942 ms



Caveat Emptor

Overview

- We're using Python here - but these strategies work well for Java too
 - For things that only apply to one language, we'll let you know





Datastore Tips

Starting Out

datastore tips

- General Strategies
 - Minimizing calls per request
 - Favor optimizing frequently viewed pages
 - **Migrate to HRD**
- Specific Examples
 - Use key ids/names
 - Use run, not fetch (python)
 - Use projection queries (for few properties)
 - Use datastore cursors
 - Use "embedded entities" to store structured data
 - Set indexed=False



Datastore Anti-Design Pattern #1

datastore tips

Don't do fetch(1):

```
query_for_my_data = MyModel.gql('WHERE user = :1', 'current_user')  
my_data = query_for_my_data.fetch(1)
```

PY

Use get (by id/key name):

```
my_data = MyModel.get_by_key_name('current_user')
```

PY



Datastore Anti-Design Pattern #1

measurements

- Before
 - 1 Datastore Fetch Op, 1 Datastore Query Op (2 Datastore Read Ops)
 - 59 ms (as measured by AppStats)
- After
 - 1 Datastore Fetch Op (1 Datastore Read Op)
 - 4 ms (as measured by AppStats)
- Summary
 - Costs **50% less** and is **14x faster**



Datastore Anti-Design Pattern #2

datastore tips

In fact, in Python, don't use "fetch" at all, really:

```
query_for_my_data = MyModel.gql('WHERE user = :1 LIMIT 100', 'current_user')
my_data = query_for_my_data.fetch(100)
for result in my_data:
    ...
```

PY

Use run:

```
query_for_my_data = MyModel.gql('WHERE user = :1', 'current_user')
my_data = query_for_my_data.run(limit=100)
for result in my_data:
    ...
```

PY



Datastore Anti-Design Pattern #3

datastore tips

When querying small entities, don't do this:

```
query_for_comments = Comments.gql('WHERE post = :1', 'post')
for comment in query_for_comments:
    print comment.text
    print comment.author
```

PY

Use Projection Queries:

```
query_for_comments = db.gql('SELECT text, author FROM Comments WHERE post = :1', 'post')
for comment in query_for_comments:
    print comment.text
    print comment.author
```

PY



Datastore Anti-Design Pattern #3

measurements

- Before
 - 1 Datastore Read Op (.07/100K ops)
 - Full entity fetch (more bandwidth, longer latency)
- After
 - 1 Datastore Small Ops (.01/100K ops)
 - Only necessary fields
- Summary
 - Costs **85% less**



Datastore Anti-Design Pattern #4

datastore tips

Don't use offsets:

```
query_for_comments = Comments.gql('WHERE post = :1 LIMIT 10 OFFSET 10', 'post')
for comment in query_for_comments:
    ...
```

PY

Use cursors:

```
cursor = self.request.get('cursor')
comments, next_cursor, more = Comments.query('post = ', 'post').fetch(10, start_cursor = cursor)
for comment in comments:
    ...
```

PY



Datastore Anti-Design Pattern #4

measurements

- Before
 - 20 Datastore Fetch Ops, 1 Datastore Query Op (21 Datastore Read Ops)
 - up to 177 ms (as measured by AppStats)
- After
 - 10 Datastore Fetch Ops, 1 Datastore Query Op (11 Datastore Read Ops)
 - 13 ms (as measured by AppStats)
- Summary
 - Costs **47% less** and is up to **13x faster**



Datastore Design Pattern #1

datastore tips

- Embedded entities allow you to store structured data within a model
- Can be helpful for keep data denormalized for more efficient querying for a page

```
class Address(ndb.Model):
    type = ndb.StringProperty() # E.g., 'home', 'work'
    street = ndb.StringProperty()
    city = ndb.StringProperty()

class Contact(ndb.Model):
    name = ndb.StringProperty()
    display_name = ndb.StringProperty()
    addresses = ndb.LocalStructuredProperty(Address, repeated=True)

new_contact = Contact(name='Marzia Niccolai',
                      display_name='Marce',
                      addresses=[Address(type='home',
                                         city='London'),
                                Address(type='work',
                                         street='Spear St',
                                         city='San Francisco')])

new_contact.put()
```

PY



Datastore Design Pattern #2

datastore tips

- If a property of a model won't be used in a query, set that property to `indexed=False`
- When you write or update the entity you won't need to write any indexes for these properties

```
class Contact(ndb.Model):
    name = ndb.StringProperty()
    display_name = ndb.StringProperty( indexed=False)
    addresses = ndb.LocalStructuredProperty(Address, repeated=True)

new_contact = Contact(name='Marzia Niccolai',
                      display_name='Marce',
                      addresses=[Address(type='home',
                                         city='London'),
                                Address(type='work',
                                         street='Spear St',
                                         city='SF')])

new_contact.put()
```

PY





Caching, Caching, Caching

Memcache

caching, caching, caching

Place frequently used and slow to compute data in memcache

```
...
greetings = memcache.get('greetings')
if greetings is not None:
    return greetings
else:
    comments = render_greetings()
    memcache.add('greetings', greetings)
    return greetings

def render_greetings():
    greetings = db.GqlQuery('SELECT text, author FROM Greetings')

    output = StringIO.StringIO()
    for greeting in greetings:
        output.write('<b>%s</b> wrote:' % greeting.author)
        output.write('<blockquote>%s</blockquote>' %
                      cgi.escape(result.content))
    return output.getvalue()
```

PY



Instance Caching

caching, caching, caching

- Allows you to have your own eviction policy for data
- Can be faster (memcache usually has around 2-5ms latency)

PY

```
GREETINGS = None
...
def get_greetings(self):
    global GREETINGS
    if GREETINGS is None:
        GREETINGS = memcache.get("greetings")
    if GREETINGS is None:
        greetings = render_greetings()
        memcache.add("greetings", greetings)
        GREETINGS = greetings
    return GREETINGS
```



But, Really, Use NDB

caching, caching, caching

- Making your caching algorithm correct is hard:

PSEUDO

```
PUT(K, V):
mc.put(K, LOCK(0), ttl=30) # Tombstone entity
db.put(K, V)
mc.delete(K) # Remove tombstone

GET(K):
v = mc.get(K)
if v is null
    r = uuid()
    was_added = mc.add(K, LOCK(r), ttl=30) #add & was_added is a minor optimization
    cas_value = was_added ? mc.get(K) : 0
    v = db.get(K)
    if case_value == LOCK(r):
        mc.cas(cas_value, v)
else if isinstance(v, LOCK)
    return db.get(K)
else
    return v
```



Datastore vs Memcache vs Instance Caching

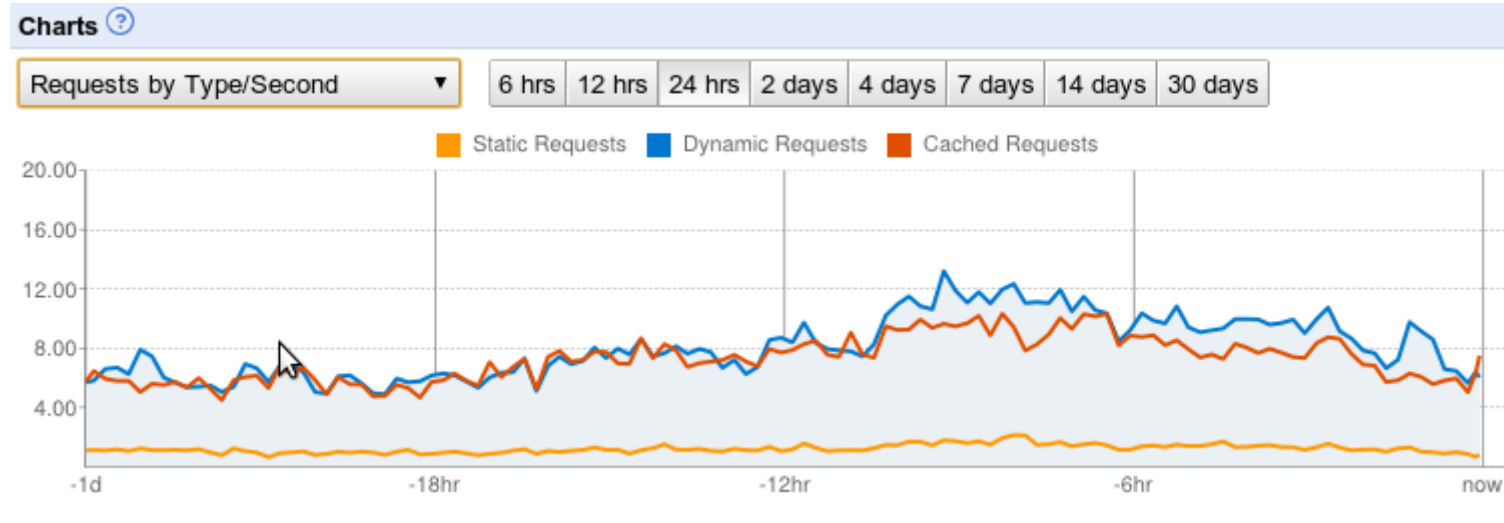
caching, caching, caching

	Datastore	Memcache	Instance Caching
Latency	50+ ms	2-5 ms	<1 ms
Total Storage	Unlimited (paid)	Limited	Limited; Instance Size Dependent
Per Entity Storage	1 MB	1 MB	Instance Size Dependent
Eviction Policy	User Managed	LRU evicted	User Managed/Instance Lifetime



Edge Caching

caching, caching, caching



```
class MyHandler(webapp.RequestHandler):  
    def get(self):  
        self.response.headers.add_header('cache-control', 'public, max-age=7200') # 2hr cache  
        self.response.out.write("<html><body><p>Hi there!</p></body></html>")
```

PY





Batch API Requests

What are batch api requests?

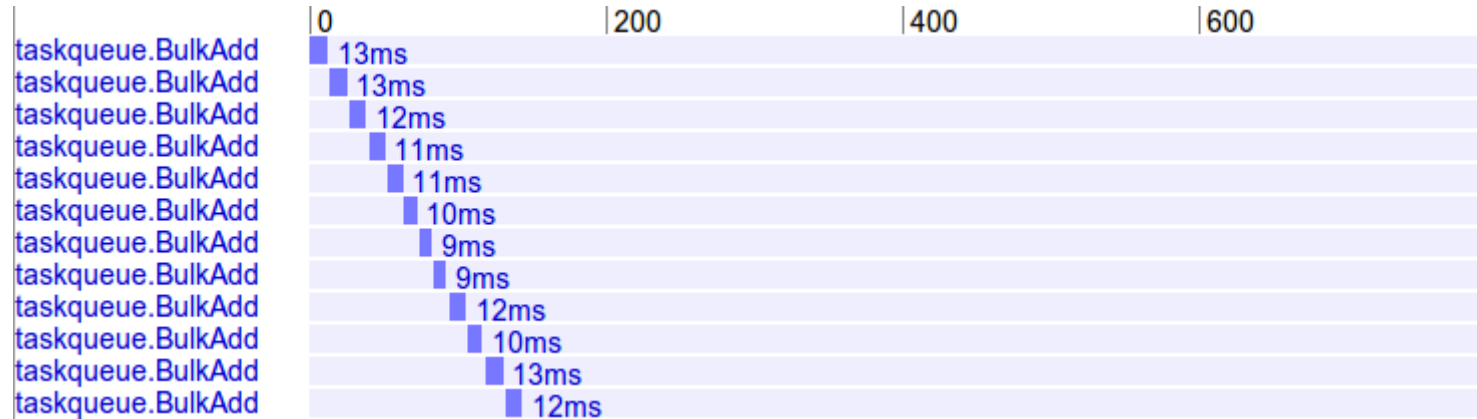
batch API requests

- Requests that operate on more than one piece of data
 - Allows requests to 'fan out' to multiple servers
 - Reduces overhead per piece of data



An example with Taskqueue

Anti-pattern



```
from google.appengine.api import taskqueue

for i in xrange(50):
    taskqueue.add(name='offline-task-%d' % i,
                  url='/my_offline_process',
                  params={'value': 1},
                  countdown=i,
                  queue_name='default')
```

PY



An example with Taskqueue

Using batch api



```
from google.appengine.api import taskqueue

queue = taskqueue.Queue('default')
tasks = []
for i in xrange(50):
    tasks.append(taskqueue.Task(name='offline-task-%d' % i,
                                url='/my_offline_process',
                                params={'value': 1},
                                countdown=i))

queue.add(tasks)
```

PY



APIs supporting batch api requests

batch API requests

- API that supports batch calls:
 - Memcache (get/set)
 - Datastore (put/get/query)
 - Taskqueue (add/lease_tasks)
 - Full Text Search (index.add)



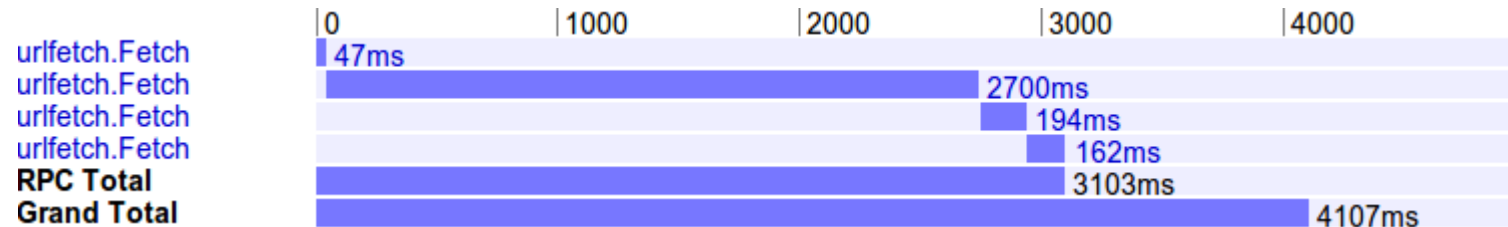


Asynchronous Workflows

Introduction to Asynchronicity

Asynchronous workflows

- When to use:
 - Perform 'slow' operations in parallel
 - Perform cpu intensive operations while waiting for IO to complete



Asynchronous URLFetch Example

Asynchronous workflows

```
from google.appengine.api import urlfetch
from google.appengine.api import apiproxy_stub_map

def event_loop(rpcs):
    while rpcs:
        rpc = apiproxy_stub_map.UserRPC.wait_any(rpcs)
        rpcs.remove(rpc)

def do_fetch(url):
    def _done_callback():
        urlfetch_response = rpc.get_result()
        print 'Fetched url', url
        # Do parsing/etc here...
    rpc = urlfetch.make_fetch_call(
        url=url, rpc=urlfetch.create_rpc(callback=_done_callback))
    return rpc

def main():
    urls = ('http://www.google.com', 'http://developers.google.com')
    _rpcs = [do_fetch(url) for url in urls]
    event_loop(_rpcs)
main()
```

PY



Asynchronous URLFetch Example (using ndb tasklets)

Asynchronous workflows

PY

```
from google.appengine.api import urlfetch
from google.appengine.ext import ndb

@ndb.tasklet
def do_fetch(url):
    urlfetch_result = yield urlfetch.make_fetch_call(
        url=url, rpc=urlfetch.create_rpc())
    # Do some processing with the urlfetch result
    print 'Fetched url', url
    raise ndb.Return(urlfetch_result)

@ndb.toplevel
def main():
    urls = ('http://www.google.com', 'http://developers.google.com')
    urlfetch_results = yield [do_fetch(url) for url in urls]
    print [len(result.content) for result in urlfetch_results]
main()
```

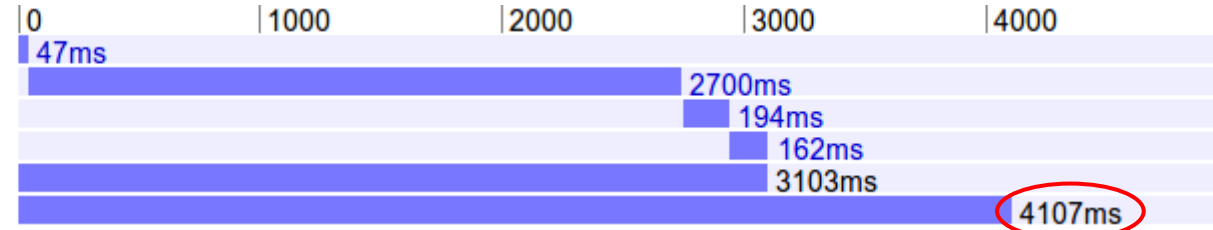


Asynchronous URLFetch Example

Asynchronous workflows

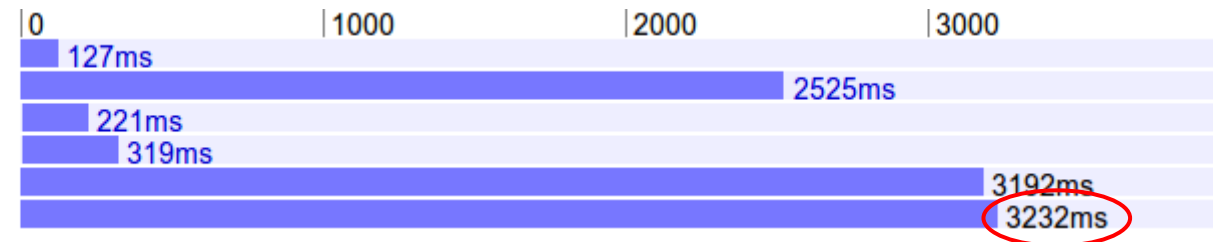
Before

urlfetch.Fetch
urlfetch.Fetch
urlfetch.Fetch
urlfetch.Fetch
RPC Total
Grand Total



After

urlfetch.Fetch
urlfetch.Fetch
urlfetch.Fetch
urlfetch.Fetch
RPC Total
Grand Total



APIs with Asynchronous Methods

Asynchronous workflows

The following APIs have asynchronous methods available

- Blobstore (create_upload_url_async, delete_async)
- Memcache (set/get/delete-multi, incr/decr, more!)
- URLFetch
- Datastore





Move Work Offline

Taskqueues

Move work offline

- If work can be deferred, then defer it
 - Updating cached values
 - Waiting for external services
- Also useful for workflow systems
- Perform fan in (Brett Slatkin, Google IO 2010: <http://goo.gl/CaGaA>)



Taskqueues

Using taskqueue to a slow external resource

```
class UrlCache(ndb.Model):
    content = ndb.BlobProperty(compressed=True)
    last_updated = ndb.DateTimeProperty(indexed=False, auto_now=True)

def _truncate_time(t, interval):
    return (t / interval) * interval

def get_url(url):
    key_name = 'x' + hashlib.sha1(url).hexdigest()
    entry = UrlCache.get_by_id(key_name)
    if not entry:
        return update_cache(url, key_name).content
    if datetime.datetime.now() - entry.last_updated > datetime.timedelta(seconds=FIVE_MINUTES):
        try:
            deferred.defer(update_cache, url, key_name,
                           _name='urlcache-%s-%d' % (key_name, _truncate_time(time.time(), FIVE_MINUTES)))
        except (taskqueue.TaskAlreadyExistsError, taskqueue.TombstonedTaskError): pass
    return entry.content

def update_cache(url, key_name):
    entry = UrlCache(content=urlfetch.fetch(url).content, id=key_name)
    entry.put()
    return entry
```

PY





Performance Settings

Performance Settings

[Responsive Search](#)

[Text Search](#)

[Datastore Admin](#)

[Memcache Viewer](#)

Administration

[Application Settings](#)

[Permissions](#)

[Blacklist](#)

[Admin Logs](#)

Billing

[Billing Settings](#)

[Billing History](#)

Resources

[Documentation](#)

[FAQ](#)

[Developer Forum](#)

[Downloads](#)

Performance

Frontend Instance Class:

Adjusting your Frontend Instance Class will affect all frontend versions of your application. Your frontends will have more memory and processing power, but also consume frontend instance hours at an increased rate, which will lead to increased costs. For example an F2 consumes instance hours at twice the rate of an F1. [Learn more.](#)

Idle Instances: (Automatic – Automatic)

The Idle Instances slider allows you to control the number of idle instances available to the default version of your application at any given time. Idle Instances are pre-loaded with your application code, so when a new Instance is needed, it can serve traffic immediately. You will not be charged for idle instances over the specified maximum. A smaller number of idle Instances means your application costs less to run, but may encounter more startup latency during load spikes. [Learn more.](#)



Pending Latency: (Automatic – Automatic)

The Pending Latency slider controls how long requests spend in the pending queue before being served by an Instance of the default version of your application. If the minimum pending latency is high App Engine will allow requests to wait rather than start new Instances to process them. This can reduce the number of instance hours your application uses, but can result in more user-visible latency. [Learn more.](#)



Save Settings



Optimizing for Low Latency

Performance Settings







- `min_idle_instances`
 - Helps handle bursty traffic
 - Avoids the dreaded Loading Request
- `max_pending_latency`
 - Keep client latencies low



Optimizing for Low Latency

Instances Console

Total number of instances	Average QPS*	Average Latency*	Average Memory
6 total (3 Resident)	0.000	Unknown ms	105.1 MBytes

Instances ?								
QPS*	Latency*	Requests	Errors	Age	Memory	Logs	Availability	Shutdown
0.000	0.0 ms	27	0	14:19:46	106.2 MBytes	View Logs	 Resident	<button>Shutdown</button>
0.000	0.0 ms	2	0	14:18:39	104.3 MBytes	View Logs	 Resident	<button>Shutdown</button>
0.000	0.0 ms	7	0	14:24:29	102.5 MBytes	View Logs	 Resident	<button>Shutdown</button>
0.000	0.0 ms	34	0	3:33:57	106.2 MBytes	View Logs	 Dynamic	<button>Shutdown</button>
0.000	0.0 ms	179	0	14:14:00	106.0 MBytes	View Logs	 Dynamic	<button>Shutdown</button>
0.000	0.0 ms	152	0	14:04:00	105.5 MBytes	View Logs	 Dynamic	<button>Shutdown</button>



Optimizing for Low Latency

Performance Settings

Idle Instances: (3 – Automatic)

The Idle Instances slider allows you to control the number of idle instances available to the default version of your application at any given time. Idle Instances are pre-loaded with your application code, so when a new Instance is needed, it can serve traffic immediately. You will not be charged for idle instances over the specified maximum. A smaller number of idle Instances means your application costs less to run, but may encounter more startup latency during load spikes. [Learn more.](#)

⚠ You currently do not have Warmup Requests enabled.

In order for your Min Idle Instances setting to immediately spin up instances you need to enable [Warmup Requests](#) for the default version of your application. You can configure Warmup Requests in your app.yaml or appengine-web.xml configuration files.



Pending Latency: (Automatic – 250ms)

The Pending Latency slider controls how long requests spend in the pending queue before being served by an Instance of the default version of your application. If the minimum pending latency is high App Engine will allow requests to wait rather than start new Instances to process them. This can reduce the number of instance hours your application uses, but can result in more user-visible latency. [Learn more.](#)

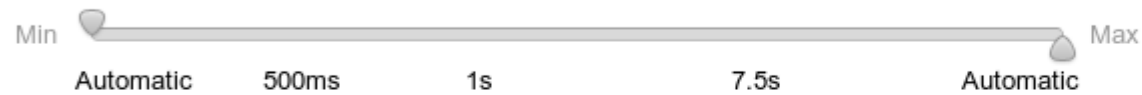


PageSpeed Service

Performance Settings

Pending Latency: (Automatic – Automatic)

The Pending Latency slider controls how long requests spend in the pending queue before being served by an Instance of the default version of your application. If the minimum pending latency is high App Engine will allow requests to wait rather than start new Instances to process them. This can reduce the number of instance hours your application uses, but can result in more user-visible latency. [Learn more.](#)



PageSpeed Service:

The PageSpeed Service automatically optimizes and caches your site for improved performance. To configure advanced settings, edit the pagespeed section in your app.yaml file.

Enable PageSpeed Service

Flush Cache

Save Settings



Optimizing for Low Cost

Performance Settings

- `max_idle_instances`
 - Keeps idle instances low to save money
 - Affects 'Frontend Instance Hours' charge
- `min_pending_latency`
 - Allows requests to wait longer for busy instances

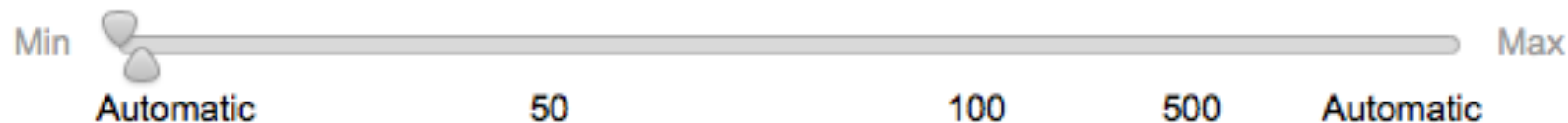


Optimizing for Low Cost

Performance Settings

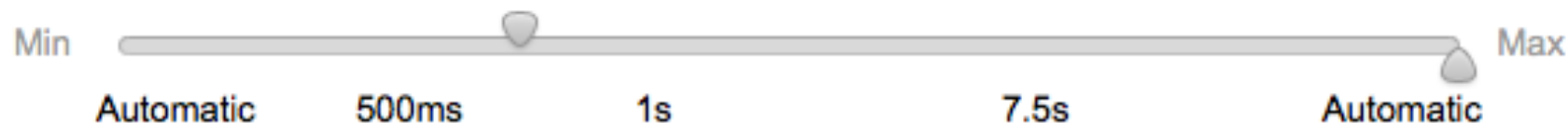
Idle Instances: (Automatic – 3)

The Idle Instances slider allows you to control the number of idle instances available to the default version of your application at any given time. Idle Instances are pre-loaded with your application code, so when a new Instance is needed, it can serve traffic immediately. You will not be charged for idle instances over the specified maximum. A smaller number of idle Instances means your application costs less to run, but may encounter more startup latency during load spikes. [Learn more.](#)



Pending Latency: (750ms – Automatic)

The Pending Latency slider controls how long requests spend in the pending queue before being served by an Instance of the default version of your application. If the minimum pending latency is high App Engine will allow requests to wait rather than start new Instances to process them. This can reduce the number of instance hours your application uses, but can result in more user-visible latency. [Learn more.](#)



Anti-Pattern

Performance Settings

Using Low Cost settings and Low Latency settings **Don't do this!**

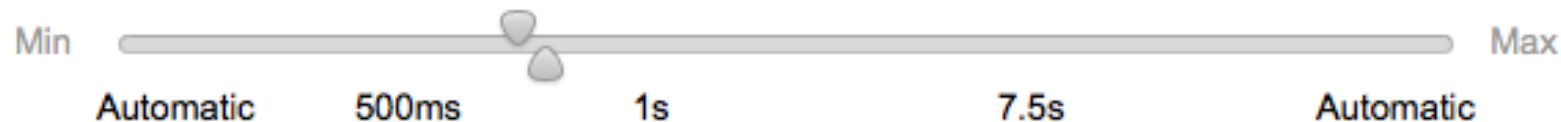
Idle Instances: (15 – 15)

The Idle Instances slider allows you to control the number of idle instances available to the default version of your application at any given time. Idle Instances are pre-loaded with your application code, so when a new Instance is needed, it can serve traffic immediately. You will not be charged for idle instances over the specified maximum. A smaller number of idle Instances means your application costs less to run, but may encounter more startup latency during load spikes. [Learn more.](#)



Pending Latency: (750ms – 800ms)

The Pending Latency slider controls how long requests spend in the pending queue before being served by an Instance of the default version of your application. If the minimum pending latency is high App Engine will allow requests to wait rather than start new Instances to process them. This can reduce the number of instance hours your application uses, but can result in more user-visible latency. [Learn more.](#)

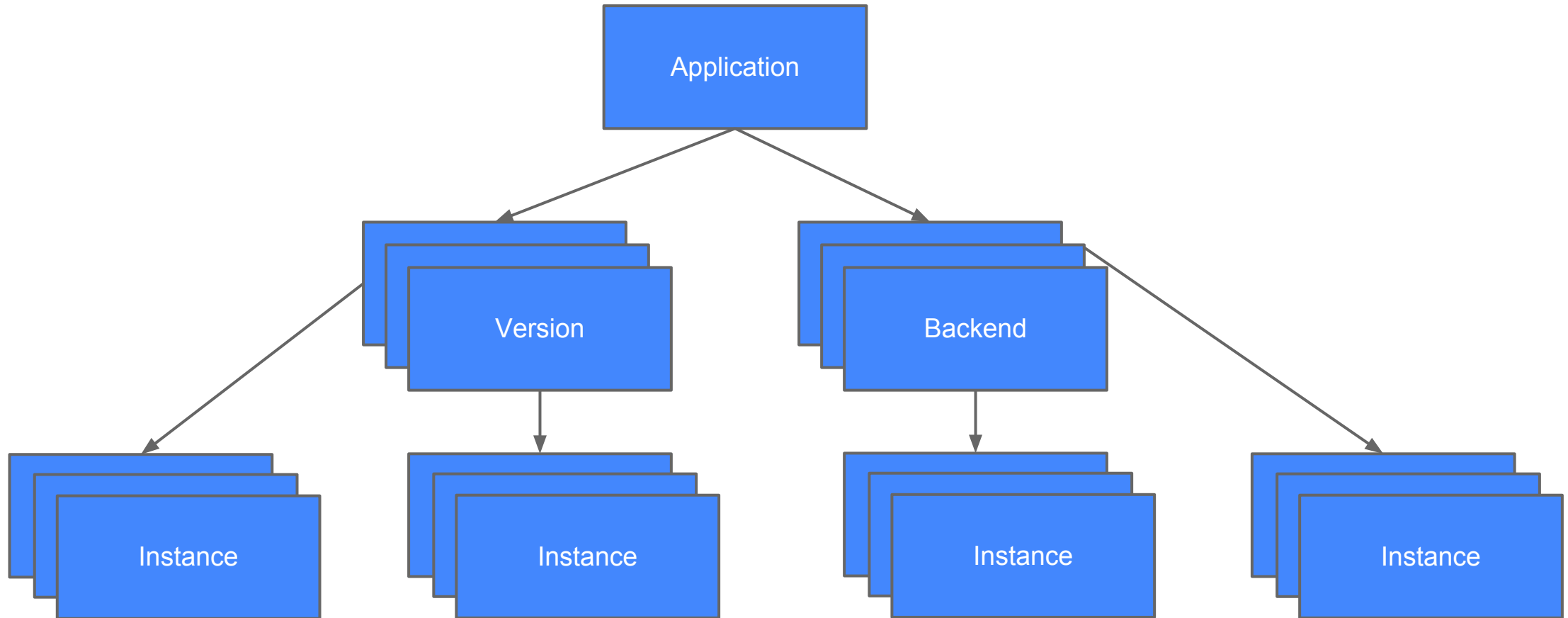




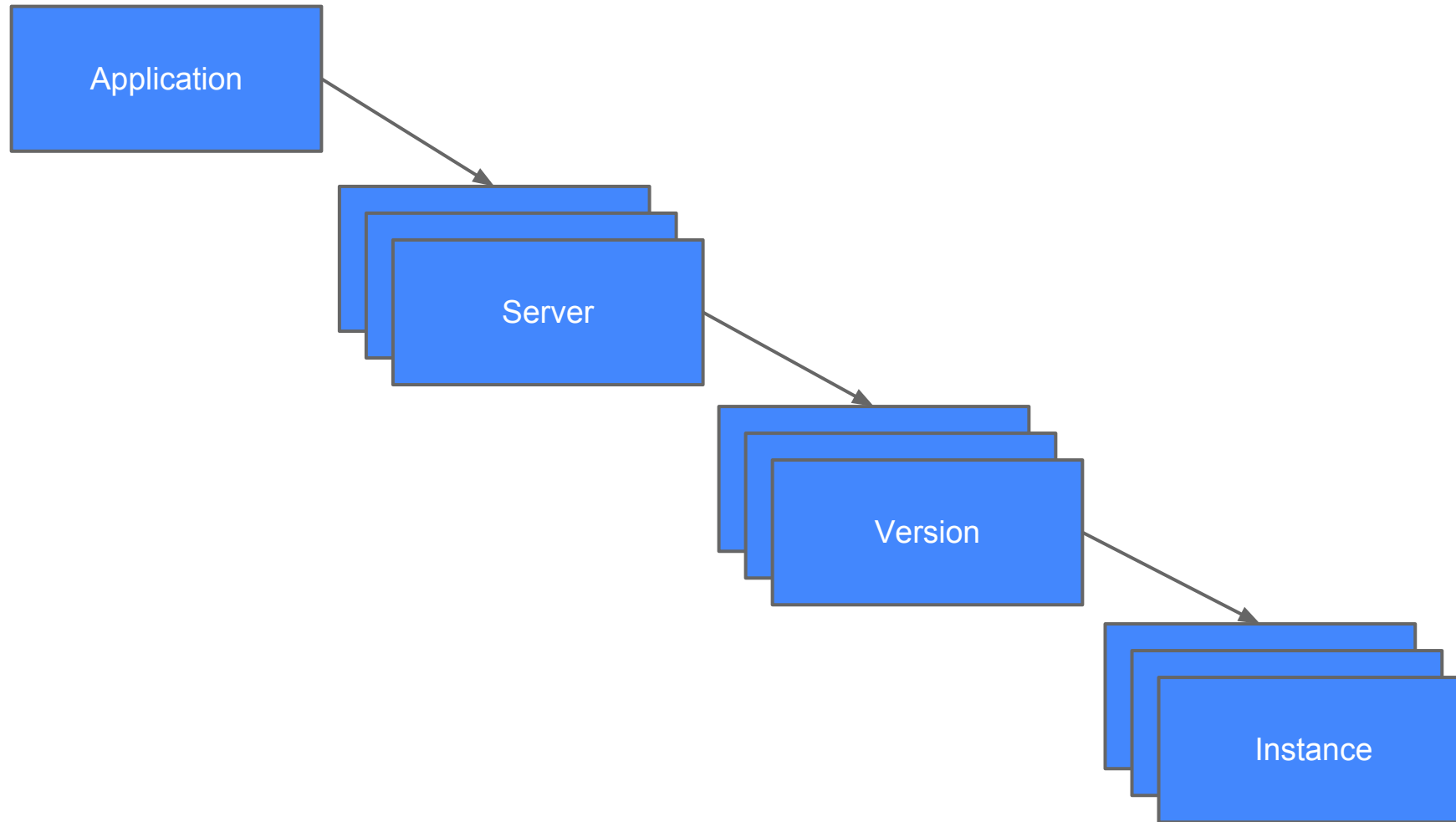
App Engine Servers

The Future...

Today's App Engine hierarchy



Tomorrow's App Engine hierarchy



App Engine Server Configuration

Mobile Frontend Server

```
application: myapp
server: mobile-fe
version: 1

runtime: python

server_settings:
  instance_class: F2
  min_idle_instances: 25
  max_idle_instances: automatic
  min_pending_latency: automatic
  max_pending_latency: 250ms

handlers:
  ...
```

YAML



App Engine Server Configuration

Geo Backend Server

```
application: myapp
server: geo-be
version: 1
runtime: java

server_settings:
  instances: 10
  instance_class: B8

handlers:
  ...
```

YAML



Recap...

- Marzia covered Datastore modeling patterns and anti-patterns
- Greg talked about batch apis, asynchronous apis and offline requests
- Troy discussed optimization through Performance Settings and the new world of Servers to come



Thank You! Questions?



Troy Trimble

Marzia Niccolai

Greg Darke