## 10. Asynchronous Processing

### 10.1 Celery Configuration

The Setarcos application uses Celery for asynchronous task processing, with optimized configurations for performance, resource efficiency, and cost-effectiveness.

#### 10.1.1 Exchange and Queue Structure

The system implements a specialized exchange and queue structure to optimize task processing:

```python

# Define exchanges

default\_exchange = Exchange('default', type='direct')

notification\_exchange = Exchange('notifications', type='direct')

analytics\_exchange = Exchange('analytics', type='direct')

rating\_exchange = Exchange('ratings', type='direct')

# Define queues with priorities

task\_queues = (

# High-priority time-sensitive queues

Queue('notifications\_critical', notification\_exchange, routing\_key='notifications.critical',

queue\_arguments={'x-max-priority': 10}),

Queue('notifications\_standard', notification\_exchange, routing\_key='notifications.standard',

queue\_arguments={'x-max-priority': 5}),

# Rating system queues

Queue('ratings\_realtime', rating\_exchange, routing\_key='ratings.realtime',

queue\_arguments={'x-max-priority': 8}),

Queue('ratings\_analysis', rating\_exchange, routing\_key='ratings.analysis',

queue\_arguments={'x-max-priority': 3}),

# Analytics queues

Queue('analytics\_realtime', analytics\_exchange, routing\_key='analytics.realtime',

queue\_arguments={'x-max-priority': 6}),

Queue('analytics\_batch', analytics\_exchange, routing\_key='analytics.batch',

queue\_arguments={'x-max-priority': 2}),

# Default queue for other tasks

Queue('default', default\_exchange, routing\_key='default',

queue\_arguments={'x-max-priority': 0}),

)

```

#### 10.1.2 Task Routing Configuration

Tasks are routed to appropriate queues based on their type and priority:

```python

task\_routes = {

# Notification tasks

'tasks.notification\_tasks.send\_push\_notification': {

'queue': 'notifications\_critical',

'routing\_key': 'notifications.critical'

},

'tasks.notification\_tasks.send\_email\_notification': {

'queue': 'notifications\_standard',

'routing\_key': 'notifications.standard'

},

'tasks.notification\_tasks.process\_notification\_queue': {

'queue': 'notifications\_standard',

'routing\_key': 'notifications.standard'

},

# Rating tasks

'tasks.rating\_tasks.submit\_rating': {

'queue': 'ratings\_realtime',

'routing\_key': 'ratings.realtime'

},

'tasks.rating\_tasks.analyze\_quality\_correlation': {

'queue': 'ratings\_analysis',

'routing\_key': 'ratings.analysis'

},

'tasks.rating\_tasks.update\_quality\_prediction\_weights': {

'queue': 'ratings\_analysis',

'routing\_key': 'ratings.analysis'

},

# Analytics tasks

'tasks.analytics\_tasks.track\_rating\_event': {

'queue': 'analytics\_realtime',

'routing\_key': 'analytics.realtime'

},

'tasks.analytics\_tasks.track\_notification\_event': {

'queue': 'analytics\_realtime',

'routing\_key': 'analytics.realtime'

},

'tasks.analytics\_tasks.generate\_engagement\_insights': {

'queue': 'analytics\_batch',

'routing\_key': 'analytics.batch'

},

# XP and gamification tasks

'tasks.xp\_tasks.calculate\_xp\_award': {

'queue': 'default',

'routing\_key': 'default'

},

'tasks.xp\_tasks.update\_user\_level': {

'queue': 'default',

'routing\_key': 'default'

},

'tasks.xp\_tasks.check\_badge\_eligibility': {

'queue': 'default',

'routing\_key': 'default'

},

}

```

#### 10.1.3 Performance and Resource Optimization

The following configurations optimize performance and resource utilization:

```python

# Time limits for tasks

task\_time\_limit = 300 # 5 minutes

task\_soft\_time\_limit = 240 # 4 minutes

# Concurrency and prefetch settings

worker\_prefetch\_multiplier = 1 # Prevent worker from prefetching too many tasks

worker\_max\_tasks\_per\_child = 1000 # Restart worker after 1000 tasks to prevent memory leaks

# Result backend configuration

result\_backend = 'redis://localhost:6379/0'

result\_expires = 60 \* 60 \* 24 # 24 hours

# Optimization settings

worker\_disable\_rate\_limits = True # Disable rate limits for better throughput

task\_acks\_late = True # Acknowledge tasks after execution (prevents lost tasks)

task\_reject\_on\_worker\_lost = True # Requeue tasks if worker dies

task\_default\_rate\_limit = '1000/m' # Default rate limit

```

#### 10.1.4 Worker Pool Configuration

Different worker pools are used based on task characteristics:

```python

# celery\_app.py

from celery import Celery

import os

app = Celery('setarcos')

app.config\_from\_object('celery\_config')

# Use gevent for I/O-bound tasks (notifications, API calls)

app.conf.worker\_pool = 'gevent'

app.conf.worker\_pool\_restarts = True

# Set concurrency based on available cores

app.conf.worker\_concurrency = os.cpu\_count() \* 2

# Enable priority support

app.conf.task\_queue\_max\_priority = 10

app.conf.task\_default\_priority = 5

```

#### 10.1.5 Specialized Worker Types

The system uses specialized workers for different task categories:

```bash

# Start notification workers (high concurrency for I/O-bound tasks)

celery -A celery\_app worker -Q notifications\_critical,notifications\_standard \

-n notification\_worker@%h -P gevent -c 50 --loglevel=info

# Start rating workers (balanced for CPU and I/O)

celery -A celery\_app worker -Q ratings\_realtime,ratings\_analysis \

-n rating\_worker@%h -P gevent -c 20 --loglevel=info

# Start analytics workers (lower priority, batch processing)

celery -A celery\_app worker -Q analytics\_realtime,analytics\_batch \

-n analytics\_worker@%h -P gevent -c 10 --loglevel=info

# Start default workers for other tasks

celery -A celery\_app worker -Q default \

-n default\_worker@%h -P prefork -c 8 --loglevel=info

```

#### 10.1.6 Autoscaling Configuration

Dynamic worker scaling based on queue depth:

```json

{

"formation": {

"notification\_worker": {

"quantity": {

"min": 2,

"max": 10

},

"size": "performance-m"

},

"rating\_worker": {

"quantity": {

"min": 1,

"max": 5

},

"size": "standard-2x"

},

"analytics\_worker": {

"quantity": {

"min": 1,

"max": 3

},

"size": "standard-1x"

},

"default\_worker": {

"quantity": {

"min": 1,

"max": 3

},

"size": "standard-1x"

}

},

"rules": [

{

"name": "notification\_worker\_scale\_up",

"formation": "notification\_worker",

"rule": {

"metric": "queue\_depth",

"op": "gt",

"target": 100,

"period": "1m"

},

"quantity": "+1"

},

{

"name": "notification\_worker\_scale\_down",

"formation": "notification\_worker",

"rule": {

"metric": "queue\_depth",

"op": "lt",

"target": 10,

"period": "5m"

},

"quantity": "-1"

}

]

}

```

### 10.2 Redis Integration

#### 10.2.1 Redis Configuration

Redis serves as both the message broker and result backend for Celery:

```python

# Redis connection settings

broker\_url = 'redis://localhost:6379/1'

result\_backend = 'redis://localhost:6379/0'

# Redis connection pool settings

broker\_pool\_limit = 100 # Maximum number of connections in the pool

broker\_connection\_timeout = 10 # Connection timeout in seconds

broker\_connection\_retry = True # Retry on connection failure

broker\_connection\_max\_retries = 3 # Maximum number of retries

```

#### 10.2.2 Redis Caching Strategy

Redis is also used for caching frequently accessed data:

```python

# Cache configuration

CACHES = {

'default': {

'BACKEND': 'django\_redis.cache.RedisCache',

'LOCATION': 'redis://localhost:6379/2',

'OPTIONS': {

'CLIENT\_CLASS': 'django\_redis.client.DefaultClient',

'PARSER\_CLASS': 'redis.connection.HiredisParser',

'SOCKET\_CONNECT\_TIMEOUT': 5,

'SOCKET\_TIMEOUT': 5,

'CONNECTION\_POOL\_KWARGS': {'max\_connections': 100},

},

'KEY\_PREFIX': 'setarcos',

'TIMEOUT': 300, # 5 minutes

},

'notification\_templates': {

'BACKEND': 'django\_redis.cache.RedisCache',

'LOCATION': 'redis://localhost:6379/3',

'TIMEOUT': 86400, # 24 hours

},

'user\_preferences': {

'BACKEND': 'django\_redis.cache.RedisCache',

'LOCATION': 'redis://localhost:6379/4',

'TIMEOUT': 3600, # 1 hour

},

'rating\_analytics': {

'BACKEND': 'django\_redis.cache.RedisCache',

'LOCATION': 'redis://localhost:6379/5',

'TIMEOUT': 1800, # 30 minutes

},

}

```

### 10.3 Task Definitions

#### 10.3.1 Notification Tasks

```python

@shared\_task(bind=True, priority=10, max\_retries=3, retry\_backoff=True)

def send\_push\_notification(self, user\_id, title, body, data=None):

"""

Send a push notification to a user with high priority.

Uses exponential backoff for retries.

"""

start\_time = time.time()

try:

db = get\_db()

user = notification\_service.get\_user\_device\_token(db, user\_id)

if not user.device\_token:

return

message = messaging.Message(

notification=messaging.Notification(title=title, body=body),

token=user.device\_token,

data=data,

android=messaging.AndroidConfig(

priority='high',

ttl=60 \* 60 \* 24, # 24 hours

),

apns=messaging.APNSConfig(

headers={'apns-priority': '10'},

)

)

response = messaging.send(message)

notification\_service.log\_notification(db, user\_id, "push", f"{title}: {body}", "sent")

# Track performance metrics

processing\_time = time.time() - start\_time

logger.info(f"Push notification sent in {processing\_time:.3f}s to user {user\_id}")

return response

except Exception as e:

notification\_service.log\_notification(db, user\_id, "push", f"{title}: {body}", "failed")

logger.error(f"Failed to send push notification: {str(e)}")

self.retry(exc=e, countdown=2 \*\* self.request.retries)

@shared\_task(bind=True, priority=5, rate\_limit='100/m')

def process\_notification\_queue(self, notifications):

"""

Process a batch of notifications efficiently.

Groups similar notifications and sends them in parallel.

"""

# Group notifications by type

push\_notifications = []

email\_notifications = []

for notification in notifications:

if notification['channel'] == 'push':

push\_notifications.append(notification)

elif notification['channel'] == 'email':

email\_notifications.append(notification)

# Create parallel tasks for each notification type

tasks = []

if push\_notifications:

push\_tasks = group(

send\_push\_notification.s(

n['user\_id'], n['title'], n['body'], n.get('data')

) for n in push\_notifications

)

tasks.append(push\_tasks)

if email\_notifications:

email\_tasks = group(

send\_email\_notification.s(

n['user\_id'], n['subject'], n['body'], n.get('template')

) for n in email\_notifications

)

tasks.append(email\_tasks)

# Execute all tasks in parallel and collect results

if tasks:

job = group(tasks)

return job.apply\_async()

return None

```

#### 10.3.2 Rating System Tasks

```python

@shared\_task(bind=True, priority=8, max\_retries=2)

def submit\_rating(self, response\_id, user\_id, rating\_value, interaction\_metrics):

"""

Process a rating submission with high priority.

"""

start\_time = time.time()

db = get\_db()

try:

# Store rating in database

rating = rating\_service.store\_rating(

db, response\_id, user\_id, rating\_value, interaction\_metrics

)

# Cache rating for quick access

cache\_key = f"rating:{response\_id}:{user\_id}"

redis\_client.setex(

cache\_key,

60 \* 60 \* 24, # 24 hours

rating\_value

)

# Trigger analytics in background

track\_rating\_event.delay(

response\_id, user\_id, rating\_value, interaction\_metrics

)

# Schedule correlation analysis with lower priority

analyze\_quality\_correlation.delay(

response\_id, rating\_value, interaction\_metrics.get('predicted\_quality\_score')

)

# Track performance

processing\_time = time.time() - start\_time

logger.info(f"Rating submitted in {processing\_time:.3f}s")

return {

"status": "success",

"rating\_id": rating.id,

"processing\_time": processing\_time

}

except Exception as e:

logger.error(f"Failed to submit rating: {str(e)}")

self.retry(exc=e)

@shared\_task(bind=True, priority=3)

def analyze\_quality\_correlation(self, response\_id, user\_rating, predicted\_quality):

"""

Analyze correlation between predicted and actual quality.

Lower priority as it's not time-critical.

"""

db = get\_db()

# Get AI response details

response = rating\_service.get\_ai\_response(db, response\_id)

# Calculate correlation

correlation = quality\_correlation\_service.analyze\_correlation(

response, user\_rating, predicted\_quality

)

# Update quality metrics

quality\_correlation\_service.update\_quality\_metrics(

db, response.philosopher\_tone, response.model\_used, correlation

)

# Check if we need to update weights

should\_update = quality\_correlation\_service.should\_update\_weights(

response.philosopher\_tone, response.model\_used

)

if should\_update:

# Schedule weight update with even lower priority

update\_quality\_prediction\_weights.delay(

response.philosopher\_tone, response.model\_used

)

return {

"status": "success",

"correlation": correlation,

"weights\_update\_scheduled": should\_update

}

```

#### 10.3.3 XP and Gamification Tasks

```python

@shared\_task(bind=True)

def calculate\_xp\_award(self, user\_id, action\_type, context=None):

"""

Calculate and award XP for user actions.

"""

db = get\_db()

xp\_service = XPService(db)

# Calculate XP based on action type and context

xp\_award = xp\_service.calculate\_xp(user\_id, action\_type, context)

# Award XP to user

if xp\_award > 0:

xp\_service.award\_xp(user\_id, xp\_award, action\_type)

# Check if user leveled up

level\_up = xp\_service.check\_level\_up(user\_id)

if level\_up:

# Trigger level up notification

send\_push\_notification.delay(

user\_id,

"Level Up!",

f"You've reached level {level\_up['new\_level']}!"

)

# Check for level-based badges

check\_badge\_eligibility.delay(user\_id, "level", level\_up['new\_level'])

return {

"user\_id": user\_id,

"action\_type": action\_type,

"xp\_awarded": xp\_award

}

```

#### 10.3.4 Analytics Tasks

```python

@shared\_task(bind=True, priority=6)

def track\_event(self, event\_name, properties, user\_id=None):

"""

Track an event in PostHog.

"""

try:

from posthog import Posthog

client = Posthog(

project\_api\_key=settings.POSTHOG\_API\_KEY,

host=settings.POSTHOG\_HOST

)

client.capture(

distinct\_id=user\_id or 'anonymous',

event=event\_name,

properties=properties

)

return {

"status": "success",

"event": event\_name,

"user\_id": user\_id

}

except Exception as e:

logger.error(f"Failed to track event: {str(e)}")

self.retry(exc=e, max\_retries=3, countdown=5)

```

### 10.4 Workflow Examples

#### 10.4.1 Notification Delivery Workflow

```python

def send\_quest\_reminder(user\_id, quest\_id):

"""

Send a reminder to a user about an active quest.

"""

# Get quest details

db = get\_db()

quest = quest\_service.get\_quest(db, quest\_id)

user = user\_service.get\_user(db, user\_id)

# Check if notification should be sent

preferences = notification\_service.get\_user\_preferences(db, user\_id)

if not preferences.get('quest\_notifications', True):

return

# Create notification content

title = "Continue Your Quest"

body = f"Continue your journey with '{quest.title}'"

data = {

"type": "quest\_reminder",

"quest\_id": str(quest\_id)

}

# Send time-sensitive notification

send\_time\_sensitive\_notification.delay(

user\_id,

"quest\_reminder",

{

"quest\_id": str(quest\_id),

"quest\_title": quest.title,

"user\_tier": user.subscription\_tier

}

)

# Track event

track\_event.delay(

"quest\_reminder\_sent",

{

"quest\_id": str(quest\_id),

"quest\_title": quest.title,

"user\_tier": user.subscription\_tier

},

user\_id

)

```

#### 10.4.2 Rating Submission and Analysis Workflow

```python

def handle\_rating\_submission(response\_id, user\_id, rating\_value, interaction\_metrics):

"""

Handle a rating submission from the frontend.

"""

# Submit rating with high priority

rating\_result = submit\_rating.delay(

response\_id, user\_id, rating\_value, interaction\_metrics

)

# Award XP for rating

calculate\_xp\_award.delay(

user\_id,

"submit\_rating",

{"response\_id": response\_id, "rating\_value": rating\_value}

)

# Send notification confirmation

send\_push\_notification.delay(

user\_id,

"Rating Received",

"Thank you for your feedback!",

{"type": "rating\_confirmation", "response\_id": response\_id}

)

return rating\_result.id

```

#### 10.4.3 Batch Processing Workflow

```python

@shared\_task(bind=True)

def process\_daily\_digest(self):

"""

Process daily digest notifications for all eligible users.

"""

db = get\_db()

# Get users eligible for daily digest

eligible\_users = notification\_service.get\_users\_eligible\_for\_daily\_digest(db)

# Group users by subscription tier

users\_by\_tier = {

"free": [],

"tier\_1": [],

"tier\_2": []

}

for user in eligible\_users:

users\_by\_tier[user.subscription\_tier].append(user.id)

# Process each tier with appropriate batch size

for tier, user\_ids in users\_by\_tier.items():

# Determine batch size based on tier

batch\_size = 100 if tier == "free" else 50

# Process in batches

for i in range(0, len(user\_ids), batch\_size):

batch = user\_ids[i:i+batch\_size]

# Create batch task

process\_daily\_digest\_batch.delay(batch, tier)

return {

"status": "scheduled",

"user\_count": len(eligible\_users)

}

```

### 10.5 Monitoring and Alerting

#### 10.5.1 Prometheus Integration

```python

from celery.signals import task\_success, task\_failure, task\_retry, worker\_ready

from prometheus\_client import Counter, Histogram, Gauge

# Prometheus metrics

task\_execution\_time = Histogram(

'task\_execution\_seconds',

'Task execution time in seconds',

['task\_name', 'queue']

)

task\_success\_counter = Counter(

'task\_success\_total',

'Number of successful task executions',

['task\_name', 'queue']

)

task\_failure\_counter = Counter(

'task\_failure\_total',

'Number of failed task executions',

['task\_name', 'queue']

)

task\_retry\_counter = Counter(

'task\_retry\_total',

'Number of task retries',

['task\_name', 'queue']

)

queue\_size = Gauge(

'queue\_size',

'Number of tasks in queue',

['queue']

)

@task\_success.connect

def task\_success\_handler(sender=None, \*\*kwargs):

task = sender.name

queue = sender.request.delivery\_info.get('routing\_key', 'unknown')

# Record success

task\_success\_counter.labels(task\_name=task, queue=queue).inc()

# Record execution time

start\_time = sender.request.get('\_\_start\_time\_\_', time.time())

execution\_time = time.time() - start\_time

task\_execution\_time.labels(task\_name=task, queue=queue).observe(execution\_time)

```

#### 10.5.2 Alerting Configuration

```python

# Alerting rules for Prometheus

alerts = [

{

"name": "HighTaskFailureRate",

"expr": "rate(task\_failure\_total[5m]) / rate(task\_success\_total[5m]) > 0.1",

"for": "5m",

"labels": {

"severity": "warning"

},

"annotations": {

"summary": "High task failure rate",

"description": "Task {{ $labels.task\_name }} has a high failure rate"

}

},

{

"name": "QueueBacklog",

"expr": "queue\_size > 1000",

"for": "5m",

"labels": {

"severity": "warning"

},

"annotations": {

"summary": "Queue backlog detected",

"description": "Queue {{ $labels.queue }} has {{ $value }} tasks waiting"

}

},

{

"name": "SlowTaskExecution",

"expr": "histogram\_quantile(0.95, sum(rate(task\_execution\_seconds\_bucket[5m])) by (task\_name, le)) > 10",

"for": "5m",

"labels": {

"severity": "warning"

},

"annotations": {

"summary": "Slow task execution",

"description": "Task {{ $labels.task\_name }} is executing slowly"

}

}

]

```

### 10.6 Best Practices

#### 10.6.1 Task Design Principles

1. \*\*Idempotency\*\*: All tasks should be designed to be idempotent (can be run multiple times without side effects)

2. \*\*Atomicity\*\*: Tasks should perform atomic operations or handle partial failures gracefully

3. \*\*Timeouts\*\*: All tasks should have appropriate timeouts to prevent worker starvation

4. \*\*Retries\*\*: Critical tasks should implement retry logic with exponential backoff

5. \*\*Monitoring\*\*: All tasks should emit appropriate logs and metrics for monitoring

#### 10.6.2 Resource Optimization

1. \*\*Worker Specialization\*\*: Use specialized workers for different task types

2. \*\*Pool Selection\*\*: Use gevent for I/O-bound tasks, prefork for CPU-bound tasks

3. \*\*Concurrency Settings\*\*: Configure appropriate concurrency based on task type

4. \*\*Memory Management\*\*: Use `worker\_max\_tasks\_per\_child` to prevent memory leaks

5. \*\*Result Expiration\*\*: Set appropriate `result\_expires` to prevent result backend bloat

#### 10.6.3 Performance Optimization

1. \*\*Task Batching\*\*: Batch similar tasks together to reduce overhead

2. \*\*Task Chunking\*\*: Split large tasks into smaller chunks for parallel processing

3. \*\*Priority Queues\*\*: Use priority queues for time-sensitive tasks

4. \*\*Caching\*\*: Cache frequently accessed data to reduce database load

5. \*\*Prefetch Limitation\*\*: Set `worker\_prefetch\_multiplier=1` to prevent worker overload