Google Cloud App Engine Debugging Guide

Table of Contents

- 1. Quick Debugging Checklist
- 2. Log Analysis and Monitoring
- 3. Common Error Scenarios
- 4. Performance Debugging
- 5. Local Development Debugging
- 6. Remote Debugging Techniques
- 7. <u>Database and External Service Issues</u>
- 8. Memory and Resource Debugging
- 9. Security and Authentication Issues
- 10. Emergency Response Procedures

Quick Debugging Checklist

Immediate Steps (First 5 minutes)

Check application status: gcloud app versions list
□ View recent logs: (gcloud app logs taillimit=100)
Check error budget: gcloud app services describe default
□ Verify instance health: (gcloud app instances list)
☐ Test basic connectivity: curl https://YOUR_PROJECT_ID.appspot.com/health

Assessment Questions

- 1. When did the issue start? Check deployment history
- 2. Is it affecting all users? Check traffic patterns
- 3. Are there recent code changes? Review git commits
- 4. **Is it environment-specific?** Compare staging vs production
- 5. Are external dependencies working? Test database/API connections

Quick Fixes to Try

•	

```
#1. Restart application instances
gcloud app versions migrate CURRENT_VERSION

#2. Scale up instances temporarily
gcloud app deploy --version=hotfix --no-promote
gcloud app services set-traffic default --splits=hotfix=100

#3. Rollback to previous version
gcloud app services set-traffic default --splits=PREVIOUS_VERSION=100

#4. Clear any cached data (if applicable)
gcloud app logs read --filter="Cache cleared" --limit=10
```

Log Analysis and Monitoring

Accessing Logs

Real-time Log Streaming

```
#Stream all logs
gcloud app logs tail

#Stream specific service logs
gcloud app logs tail --service=api --version=v2

#Stream with severity filtering
gcloud app logs tail --filter="severity >= ERROR"

#Stream specific time range
gcloud app logs tail --filter='timestamp >= "2024-08-24T10:00:00Z"
```

Historical Log Analysis

bash			

```
# Get logs from last hour
gcloud app logs read --filter='timestamp >= "2024-08-24T09:00:00Z"' --limit=500

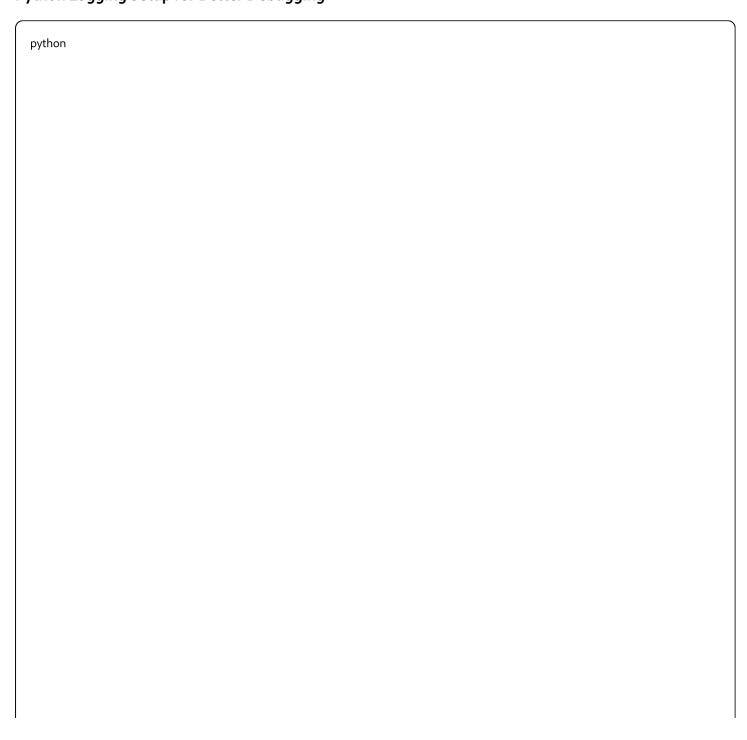
# Search for specific errors
gcloud app logs read --filter='textPayload:"ConnectionError"' --limit=100

# Filter by HTTP status codes
gcloud app logs read --filter='httpRequest.status >= 500' --limit=50

# Get logs for specific user session
gcloud app logs read --filter='trace:"projects/PROJECT_ID/traces/TRACE_ID"'
```

Log Parsing and Analysis

Python Logging Setup for Better Debugging



```
import logging
import json
import traceback
from datetime import datetime
import google.cloud.logging
# Setup structured logging
def setup_logging():
  client = google.cloud.logging.Client()
  client.setup_logging()
  # Custom formatter for structured logs
  class StructuredMessage:
   def __init__(self, message, **kwargs):
     self.message = message
     self.kwargs = kwargs
   def __str__(self):
     return json.dumps({
       'message': self.message,
       'timestamp': datetime.utcnow().isoformat(),
       **self.kwargs
     })
  return StructuredMessage
# Usage example
sm = setup_logging()
logging.info(sm("User action", user_id=123, action="login", ip="192.168.1.1"))
logging.error(sm("Database error", error=str(e), table="users", query_time=0.5))
```

Log Aggregation Queries

bash

```
#Count errors by type
gcloud logging read 'resource.type="gae_app" AND severity >= "ERROR"' \
--format="value(jsonPayload.message)" | sort | uniq -c | sort -nr

#Find slowest requests
gcloud logging read 'resource.type="gae_app" AND httpRequest.latency > "5s"' \
--format="table(timestamp, httpRequest.requestUrl, httpRequest.latency)"

#Memory usage patterns
gcloud logging read 'resource.type="gae_app" AND textPayload:"memory"' \
--format="csv(timestamp,resource.labels.version_id,jsonPayload.memory_mb)"
```

Setting Up Custom Dashboards

Error Rate Dashboard

```
# monitoring-dashboard.yaml
displayName: "App Engine Debug Dashboard"
gridLayout:
widgets:
- title: "Error Rate"
xyChart:
dataSets:
- timeSeriesQuery:
timeSeriesFilter:
filter: 'resource.type="gae_app"'
aggregation:
alignmentPeriod: "60s"
perSeriesAligner: "ALIGN_RATE"
```

Performance Metrics

python			

```
# Custom metrics for debugging
from google.cloud import monitoring_v3
import time
def record_debug_metric(metric_name, value, labels=None):
 client = monitoring_v3.MetricServiceClient()
 project_name = f"projects/{PROJECT_ID}"
 series = monitoring_v3.TimeSeries()
 series.metric.type = f"custom.googleapis.com/{metric_name}"
 series.resource.type = "gae_app"
 if labels:
   for key, val in labels.items():
     series.metric.labels[key] = str(val)
 point = series.points.add()
 point.value.double_value = value
 point.interval.end_time.seconds = int(time.time())
 client.create_time_series(name=project_name, time_series=[series])
# Usage
record_debug_metric("database_query_time", 0.25, {"table": "users", "operation": "select"})
record_debug_metric("cache_hit_rate", 0.85, {"cache_type": "redis"})
```

Common Error Scenarios

HTTP 500 Internal Server Error

Diagnosis Steps

```
#1. Check recent error logs
gcloud app logs read --filter='httpRequest.status = 500' --limit=20

#2. Look for Python tracebacks
gcloud app logs read --filter='textPayload:"Traceback"' --limit=10

#3. Check for dependency issues
gcloud app logs read --filter='textPayload:"ImportError" OR textPayload:"ModuleNotFoundError"
```

```
python
# Issue: Unhandled exceptions
# Solution: Add comprehensive error handling
from flask import Flask, jsonify
import logging
import traceback
app = Flask(__name___)
@app.errorhandler(500)
def internal_error(error):
  error_id = str(uuid.uuid4())
  logging.error(f"Internal error {error_id}: {str(error)}")
  logging.error(f"Traceback {error_id}: {traceback.format_exc()}")
  return jsonify({
   'error': 'Internal server error',
   'error_id': error_id
 }), 500
@app.route('/api/users')
def get_users():
 try:
    # Your code here
   users = get_users_from_db()
   return jsonify(users)
  except Exception as e:
   logging.error(f"Failed to get users: {str(e)}")
    logging.error(f"Traceback: {traceback.format_exc()}")
    raise
```

HTTP 502 Bad Gateway

Diagnosis

```
# Check if instances are starting properly
gcloud app instances list --filter="status != RUNNING"

# Look for startup errors
gcloud app logs read --filter='textPayload:"main.py" AND severity >= "ERROR"'

# Check health check endpoints
curl -I https://YOUR_PROJECT_ID.appspot.com/health
```

Solutions

```
yaml

# app.yaml - Add proper health checks
liveness_check:
path: "/health"
check_interval_sec: 30
timeout_sec: 4
failure_threshold: 2

readiness_check:
path: "/readiness"
check_interval_sec: 5
timeout_sec: 4
```

```
python
# Health check endpoints
@app.route('/health')
def health_check():
 try:
   # Test critical dependencies
   db.execute("SELECT 1")
   cache.get("health_check")
   return {"status": "healthy", "timestamp": datetime.utcnow().isoformat()}
  except Exception as e:
   logging.error(f"Health check failed: {str(e)}")
   return {"status": "unhealthy", "error": str(e)}, 503
@app.route('/readiness')
def readiness_check():
  # Check if app is ready to receive traffic
 if not app_initialized:
   return {"status": "not ready"}, 503
  return {"status": "ready"}
```

HTTP 404 Not Found

URL Routing Debug

bash			

```
# Check URL patterns in logs
gcloud app logs read --filter='httpRequest.status = 404' --limit=50 \
--format="table(timestamp, httpRequest.requestUrl, httpRequest.userAgent)"

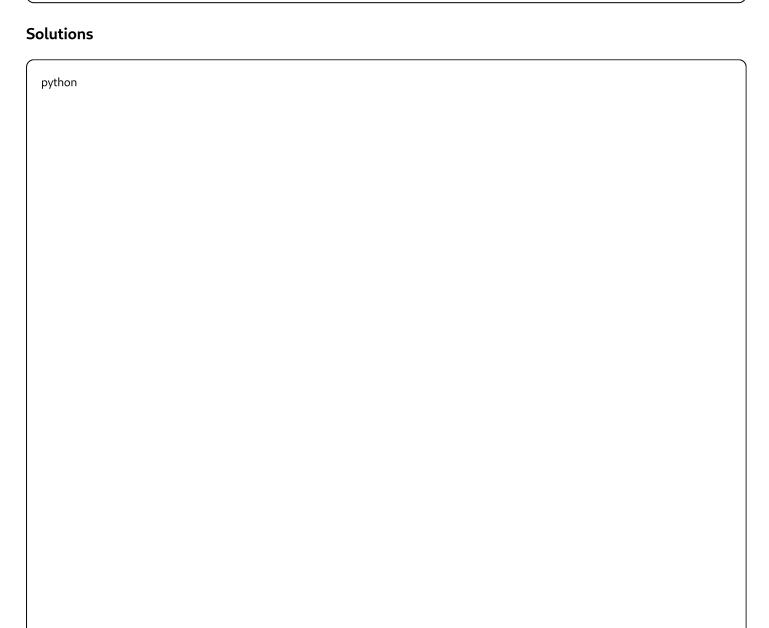
# Analyze routing configuration
gcloud app describe --service=default
```

Timeout Errors

Diagnosis

```
# Find long-running requests
gcloud app logs read --filter='httpRequest.latency > "30s"' --limit=20

# Check for DeadlineExceededError
gcloud app logs read --filter='textPayload:"DeadlineExceededError"
```



```
# Implement timeout handling
import signal
from contextlib import contextmanager
@contextmanager
def timeout_handler(seconds):
  def timeout_function(signum, frame):
   raise TimeoutError(f"Operation timed out after {seconds} seconds")
  old_handler = signal.signal(signal.SIGALRM, timeout_function)
  signal.alarm(seconds)
  try:
   yield
  finally:
   signal.alarm(0)
   signal.signal(signal.SIGALRM, old_handler)
# Usage
try:
 with timeout_handler(25): #App Engine has 30s limit
   result = slow_database_operation()
except TimeoutError as e:
 logging.error(f"Operation timed out: {str(e)}")
  return {"error": "Request timed out"}, 408
```

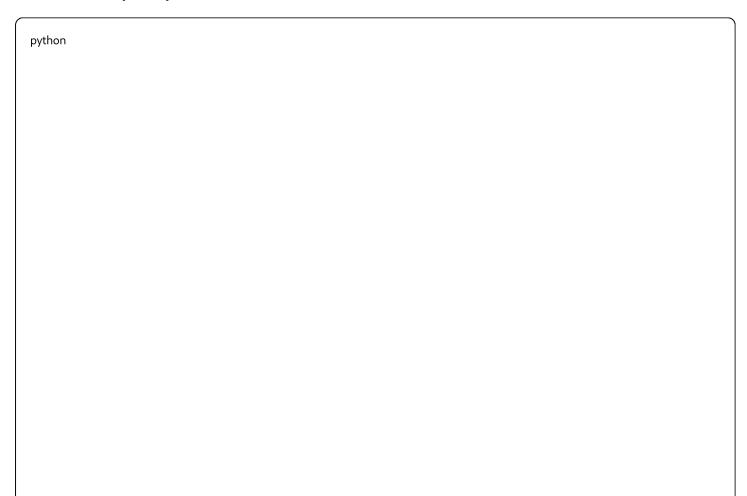
Performance Debugging

Identifying Performance Bottlenecks

Request Tracing

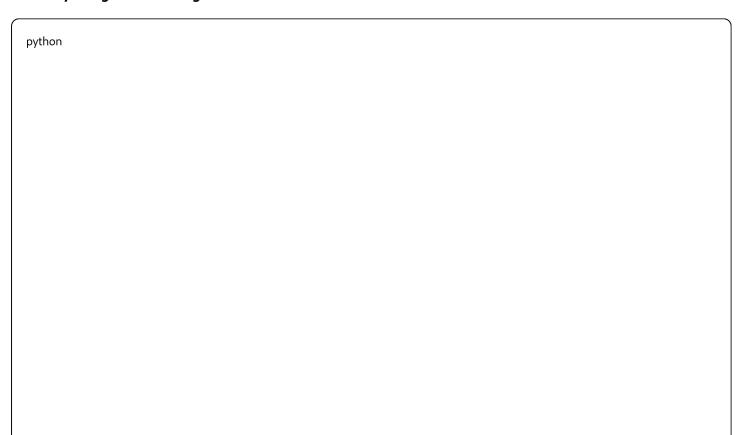
```
# Add request tracing
import time
import uuid
from flask import request, g
import logging
@app.before_request
def before_request():
 g.request_id = str(uuid.uuid4())
 g.start_time = time.time()
 logging.info(f"Request {g.request_id} started: {request.method} {request.path}")
@app.after_request
def after_request(response):
  duration = time.time() - g.start_time
 logging.info(f"Request_id} completed in {duration:.3f}s with status {response.status_code}")
  # Log slow requests
 if duration > 1.0:
   logging.warning(f"Slow request {g.request_id}: {duration:.3f}s for {request.path}")
  return response
```

Database Query Analysis



```
# Database query profiling
import functools
import time
import logging
def profile_query(func):
  @functools.wraps(func)
  def wrapper(*args, **kwargs):
   start_time = time.time()
   result = func(*args, **kwargs)
   duration = time.time() - start_time
   logging.info(f"Query executed in {duration:.3f}s: {func.__name___}")
   if duration > 0.5: #Log slow queries
     logging.warning(f"Slow query detected: {func.__name__} took {duration:.3f}s")
   return result
  return wrapper
# Usage
@profile_query
def get_user_data(user_id):
  return db.execute("SELECT * FROM users WHERE id = %s", [user_id])
```

Memory Usage Monitoring



```
# Memory usage tracking
import psutil
import gc
import logging
def log_memory_usage(context=""):
  process = psutil.Process()
  memory_info = process.memory_info()
  memory_mb = memory_info.rss / 1024 / 1024
 logging.info(f"Memory usage {context}: {memory_mb:.2f} MB")
 if memory_mb > 400: # Warning if approaching App Engine limits
   logging.warning(f"High memory usage detected: {memory_mb:.2f} MB")
    # Force garbage collection
   collected = gc.collect()
   logging.info(f"Garbage collection freed {collected} objects")
# Use at critical points
@app.route('/api/heavy-operation')
def heavy_operation():
 log_memory_usage("before operation")
  # Your heavy operation
  result = process_large_dataset()
 log_memory_usage("after operation")
  return result
```

Performance Optimization Commands

Instance Scaling Analysis

```
# Check scaling events
gcloud app logs read --filter='textPayload:"scaling"' --limit=50

# Monitor instance count over time
gcloud app instances list --format="table(service, version, id, vmStatus)"

# Check CPU and memory utilization
gcloud app operations list --filter="target.name:apps/PROJECT_ID"
```

Traffic Analysis

```
# Analyze request patterns
gcloud app logs read --filter='httpRequest.requestUrl:"/api/"'\
--format="csv(timestamp,httpRequest.requestUrl,httpRequest.latency)"\
--limit=1000 > requests.csv

# Find peak traffic times
gcloud app logs read --filter='severity="INFO"'\
--format="value(timestamp.hour())" | sort | uniq -c
```

Local Development Debugging

Setting Up Local Debug Environment

Development Server with Debugging

```
# Install development dependencies
pip install flask-debugtoolbar
pip install werkzeug

# Set up debug environment
export FLASK_ENV=development
export FLASK_DEBUG=1
export GOOGLE_CLOUD_PROJECT=your-project-id
export GOOGLE_APPLICATION_CREDENTIALS=path/to/service-key.json

# Run with App Engine dev server
dev_appserver.py --enable_console --show_mail_body app.yaml
```

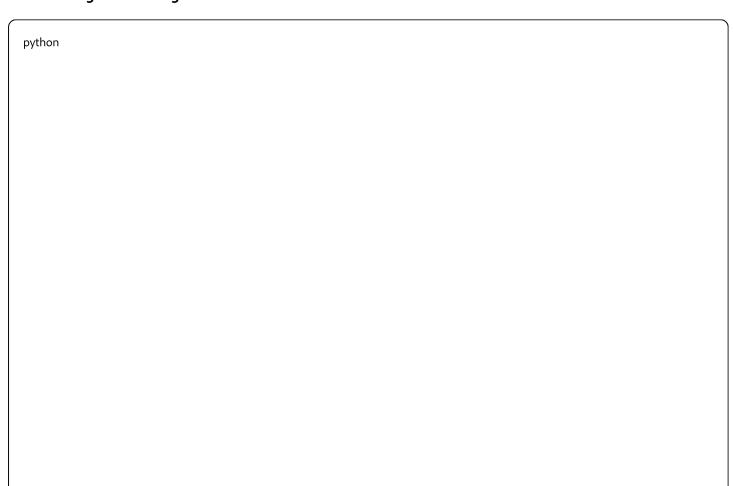
Debug Configuration

python				

```
# Debug-enabled Flask app
from flask import Flask
from\ flask\_debugtoolbar\ import\ DebugToolbarExtension
import os
app = Flask(__name__)
if os.getenv('FLASK_ENV') == 'development':
  app.debug = True
 app.config['SECRET_KEY'] = 'dev-secret-key'
 app.config['DEBUG_TB_INTERCEPT_REDIRECTS'] = False
 toolbar = DebugToolbarExtension(app)
# Enhanced logging for development
if app.debug:
 import logging
 logging.basicConfig(
   level=logging.DEBUG,
   format='%(asctime)s %(levelname)s %(name)s %(message)s'
```

Local Testing Strategies

Unit Testing with Debug Info



```
import unittest
import logging
from unittest.mock import patch, MagicMock
class TestUserAPI(unittest.TestCase):
  def setUp(self):
   self.app = create_app(testing=True)
   self.client = self.app.test_client()
   # Enable debug logging in tests
   logging.basicConfig(level=logging.DEBUG)
  def test_get_user_with_debug(self):
    # Mock external dependencies
   with patch('app.database.get_user') as mock_db:
     mock_db.return_value = {'id': 1, 'name': 'Test User'}
     response = self.client.get('/api/users/1')
      # Debug assertions
     self.assertEqual(response.status_code, 200)
     self.assertTrue(mock_db.called)
     # Log request details for debugging
     logging.debug(f"Response: {response.data}")
     logging.debug(f"Headers: {response.headers}")
if __name__ == '__main___':
  unittest.main(verbosity=2)
```

Integration Testing

```
# Integration test with real dependencies
import requests
import time
def test_full_stack():
  base_url = "http://localhost:8080"
  # Test health endpoint
  health_response = requests.get(f"{base_url}/health")
  print(f"Health check: {health_response.status_code}")
  # Test main functionality
  start_time = time.time()
  api_response = requests.get(f"{base_url}/api/users")
  end_time = time.time()
  print(f"API response time: {end_time - start_time:.3f}s")
  print(f"Response status: {api_response.status_code}")
  print(f"Response size: {len(api_response.content)} bytes")
  # Test error conditions
  error_response = requests.get(f"{base_url}/api/nonexistent")
  print(f"Error handling: {error_response.status_code}")
```

Remote Debugging Techniques

SSH Access to Instances

Connecting to Running Instances

```
# List running instances
gcloud app instances list

# SSH into specific instance
gcloud app instances ssh INSTANCE_ID --service=SERVICE --version=VERSION

# Execute commands on instance
gcloud app instances ssh INSTANCE_ID --service=SERVICE --version=VERSION \
--command="ps aux | grep python"
```

```
# Install debugging tools on instance
gcloud app instances ssh INSTANCE_ID --service=SERVICE --version=VERSION \
--command="pip install remote-pdb"

# Set up remote debugger in code
import remote_pdb
remote_pdb.set_trace(host='0.0.0.0', port=4444)

# Connect from local machine
telnet INSTANCE_EXTERNAL_IP 4444
```

Cloud Shell Debugging

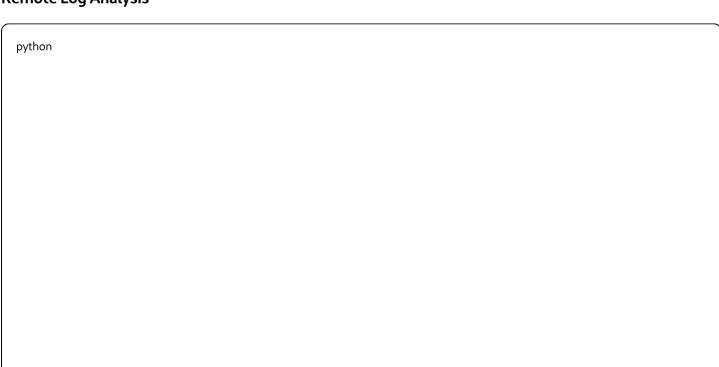
Using Cloud Shell for Investigation

```
# Start Cloud Shell session
gcloud cloud-shell ssh

# Install debugging tools
pip install --user google-cloud-logging google-cloud-monitoring

# Run debugging scripts
python debug_script.py
```

Remote Log Analysis



```
# debug_script.py - Run in Cloud Shell
from google.cloud import logging
from google.cloud import monitoring_v3
from datetime import datetime, timedelta
import json
def analyze_errors():
  client = logging.Client()
  # Get errors from last hour
  end_time = datetime.utcnow()
  start_time = end_time - timedelta(hours=1)
  filter_str = f'''
   resource.type="gae_app"
   AND severity >= "ERROR"
   AND timestamp >= "{start_time.isoformat()}Z"
  entries = client.list_entries(filter_=filter_str, order_by=logging.DESCENDING)
  error_counts = {}
  for entry in entries:
    error_type = entry.payload.get('message', 'Unknown')[:50]
    error_counts[error_type] = error_counts.get(error_type, 0) + 1
  print("Error summary:")
  for error, count in sorted(error_counts.items(), key=lambda x: x[1], reverse=True):
    print(f" {count}: {error}")
if __name__ == "__main__":
  analyze_errors()
```

Database and External Service Issues

Database Connection Debugging

Connection Pool Monitoring

```
import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool
import logging
# Enhanced database setup with debugging
def create_db_engine():
  engine = create_engine(
   DATABASE_URL,
   poolclass=QueuePool,
   pool_size=5,
   max_overflow=10,
   pool_pre_ping=True, # Verify connections before use
   echo=True if os.getenv('DEBUG_SQL') else False
  # Monitor connection pool
  @sqlalchemy.event.listens_for(engine, "connect")
  def receive_connect(dbapi_connection, connection_record):
   logging.info("Database connection established")
  @sqlalchemy.event.listens_for(engine, "checkout")
  def receive_checkout(dbapi_connection, connection_record, connection_proxy):
   logging.debug("Connection checked out from pool")
  return engine
```

Database Query Debugging

```
# Query debugging decorator
def debug_query(query_name):
 def decorator(func):
   def wrapper(*args, **kwargs):
     start_time = time.time()
     try:
       result = func(*args, **kwargs)
       duration = time.time() - start_time
       logging.info(f"Query {query_name} completed in {duration:.3f}s")
       return result
     except Exception as e:
       duration = time.time() - start_time
       logging.error(f"Query {query_name} failed after {duration:.3f}s: {str(e)}")
       raise
   return wrapper
 return decorator
@debug_query("get_user_orders")
def get_user_orders(user_id):
 with engine.connect() as conn:
   result = conn.execute(
     "SELECT * FROM orders WHERE user_id = %s ORDER BY created_at DESC",
     [user_id]
   return result.fetchall()
```

External API Debugging

HTTP Client with Debugging

```
import requests
import logging
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry
class DebugHTTPAdapter(HTTPAdapter):
  def send(self, request, **kwargs):
   logging.debug(f"Sending {request.method} request to {request.url}")
   logging.debug(f"Headers: {request.headers}")
   start_time = time.time()
   response = super().send(request, **kwargs)
   duration = time.time() - start_time
   logging.info(f"HTTP {request.method} {request.url} -> {response.status_code} ({duration:.3f}s)")
   if response.status_code >= 400:
     logging.error(f"HTTP Error: {response.status_code} {response.text}")
   return response
def create_debug_session():
  session = requests.Session()
  # Add retry strategy
  retry_strategy = Retry(
   total=3,
   backoff_factor=1,
   status_forcelist=[429, 500, 502, 503, 504],
  adapter = DebugHTTPAdapter(max_retries=retry_strategy)
  session.mount("http://", adapter)
  session.mount("https://", adapter)
  return session
# Usage
api_session = create_debug_session()
response = api_session.get("https://api.example.com/data", timeout=30)
```

Service Health Monitoring

```
# External service health checker
def check_external_services():
  services = [
   {"name": "Database", "url": DATABASE_URL, "timeout": 5},
   {"name": "Redis", "url": f"redis://{REDIS_HOST}:{REDIS_PORT}", "timeout": 3},
   {"name": "External API", "url": EXTERNAL_API_URL, "timeout": 10}
  health_status = {}
  for service in services:
   try:
     if service["name"] == "Database":
       # Test database connection
       with engine.connect() as conn:
         conn.execute("SELECT 1")
       health_status[service["name"]] = "healthy"
     elif service["name"] == "Redis":
       # Test Redis connection
       import redis
       r = redis.Redis.from_url(service["url"])
       r.ping()
       health_status[service["name"]] = "healthy"
     else:
       # Test HTTP service
       response = requests.get(service["url"], timeout=service["timeout"])
       health_status[service["name"]] = "healthy" if response.status_code == 200 else "degraded"
   except Exception as e:
     logging.error(f"Service {service['name']} health check failed: {str(e)}")
     health_status[service["name"]] = "unhealthy"
  return health_status
@app.route('/debug/services')
def debug_services():
  return jsonify(check_external_services())
```

Memory and Resource Debugging

Memory Leak Detection

Memory Usage Monitoring

```
python
import psutil
import gc
import sys
import tracemalloc
# Start memory tracing
tracemalloc.start()
class MemoryMonitor:
 def __init__(self):
   self.snapshots = []
  def take_snapshot(self, label):
   snapshot = tracemalloc.take_snapshot()
   self.snapshots.append((label, snapshot))
    # Log current memory usage
   process = psutil.Process()
   memory_mb = process.memory_info().rss / 1024 / 1024
   logging.info(f"Memory snapshot '{label}': {memory_mb:.2f} MB")
  def compare_snapshots(self, label1, label2):
   snap1 = next(s[1] for s in self.snapshots if s[0] == label1)
   snap2 = next(s[1] for s in self.snapshots if s[0] == label2)
   top_stats = snap2.compare_to(snap1, 'lineno')
   print(f"Memory diff between '{label1}' and '{label2}':")
   for stat in top_stats[:10]:
     print(stat)
# Usage
memory_monitor = MemoryMonitor()
@app.before_request
def before_request():
  memory_monitor.take_snapshot(f"request_start_{request.path}")
@app.after_request
def after_request(response):
  memory_monitor.take_snapshot(f"request_end_{request.path}")
  return response
```

Garbage Collection Analysis

```
python
import gc
import weakref
def analyze_memory():
  # Force garbage collection
  collected = gc.collect()
  logging.info(f"Garbage collection freed {collected} objects")
  # Count objects by type
  object_counts = {}
  for obj in gc.get_objects():
   obj_type = type(obj).__name__
   object_counts[obj_type] = object_counts.get(obj_type, 0) + 1
  # Log top object types
  top_objects = sorted(object_counts.items(), key=lambda x: x[1], reverse=True)[:10]
  logging.info("Top object types in memory:")
  for obj_type, count in top_objects:
   logging.info(f" {obj_type}: {count}")
  # Check for circular references
 if gc.garbage:
   logging.warning(f"Found {len(gc.garbage)} uncollectable objects")
# Schedule periodic memory analysis
from threading import Timer
def periodic_memory_check():
 analyze_memory()
 Timer(300.0, periodic_memory_check).start() # Every 5 minutes
periodic_memory_check()
```

Resource Usage Optimization

File Handle Monitoring

python			

```
import resource
import logging
def check_resource_limits():
  # Get current resource usage
 usage = resource.getrusage(resource.RUSAGE_SELF)
  logging.info(f"CPU time: {usage.ru_utime + usage.ru_stime:.2f}s")
  logging.info(f"Max memory: {usage.ru_maxrss / 1024:.2f} MB")
  logging.info(f"Page faults: {usage.ru_majflt}")
  # Check file descriptor usage
 import os
  open_fds = len(os.listdir('/proc/self/fd'))
 logging.info(f"Open file descriptors: {open_fds}")
 if open_fds > 100:
   logging.warning("High number of open file descriptors detected")
@app.route('/debug/resources')
def debug_resources():
  check_resource_limits()
 return {"status": "Resource check completed"}
```

Security and Authentication Issues

Authentication Debugging

JWT Token Validation

```
import jwt
import logging
from functools import wraps
def debug_jwt_auth(f):
  @wraps(f)
  def decorated_function(*args, **kwargs):
    token = request.headers.get('Authorization')
    if not token:
     logging.warning("No authorization token provided")
     return {"error": "No token provided"}, 401
    try:
      # Remove 'Bearer ' prefix
     if token.startswith('Bearer'):
       token = token[7:]
      # Decode token with debugging
     payload = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
     logging.info(f"Token decoded successfully for user {payload.get('user_id')}")
      # Check token expiration
     import time
     if payload.get('exp', 0) < time.time():</pre>
       logging.warning("Expired token used")
       return {"error": "Token expired"}, 401
    except jwt.InvalidTokenError as e:
     logging.error(f"Invalid token: {str(e)}")
     return {"error": "Invalid token"}, 401
    return f(*args, **kwargs)
  return decorated function
```

OAuth Flow Debugging

```
import requests
import logging
def debug_oauth_callback():
  code = request.args.get('code')
  state = request.args.get('state')
  logging.info(f"OAuth callback received - Code: {code[:10]}..., State: {state}")
  if not code:
    logging.error("No authorization code received")
    return {"error": "No authorization code"}, 400
  # Exchange code for token
  try:
   token_response = requests.post(
      OAUTH_TOKEN_URL,
     data={
       'client_id': CLIENT_ID,
       'client_secret': CLIENT_SECRET,
       'code': code,
       'grant_type': 'authorization_code'
     },
     timeout=30
    logging.info(f"Token exchange response: {token_response.status_code}")
    if token_response.status_code != 200:
     logging.error(f"Token exchange failed: {token_response.text}")
     return {"error": "Token exchange failed"}, 400
    return token_response.json()
  except requests.RequestException as e:
   logging.error(f"OAuth token exchange error: {str(e)}")
    return {"error": "Authentication failed"}, 500
```

Security Headers and CORS

CORS Debugging

```
from flask_cors import CORS
import logging

def setup_cors_with_debugging(app):
    def log_cors_request():
        origin = request.headers.get('Origin')
        method = request.headers.get('Access-Control-Request-Method')
        headers = request.headers.get('Access-Control-Request-Headers')

    logging.info(f"CORS preflight - Origin: {origin}, Method: {method}, Headers: {headers}")

@app.before_request
    def before_cors_request():
    if request.method ==
```