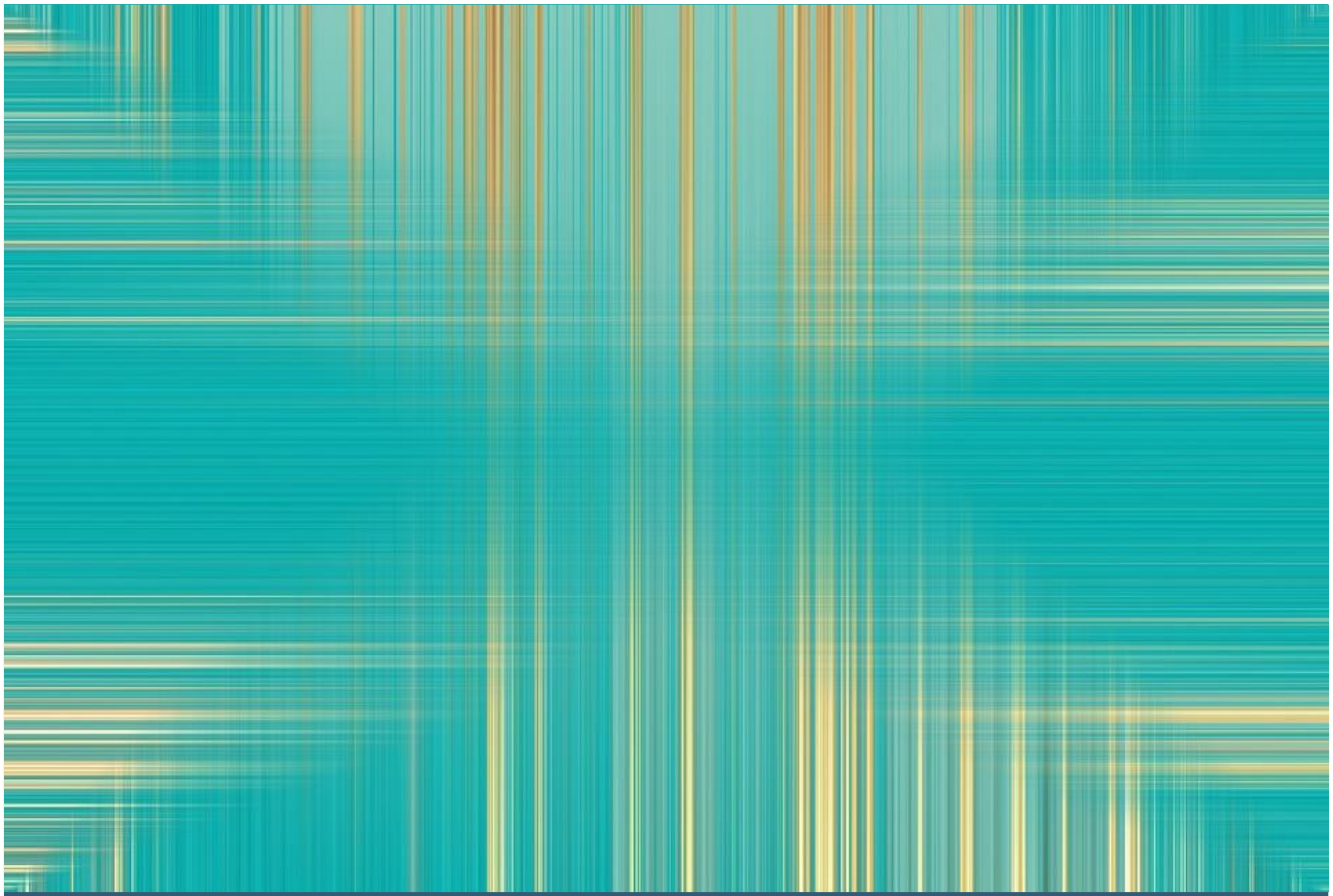


Parallel and Distributed Atom-Effect Calculator for a 3D Cartesian Grid

Stela Kucek

12/16/20

Cloud Computing

Table of Contents

1 Application Description

The designed application is a parallel and distributed atom-effect calculator within a 3D Cartesian grid, which calculates the physical effect of all atoms present in the grid space on the grid points:

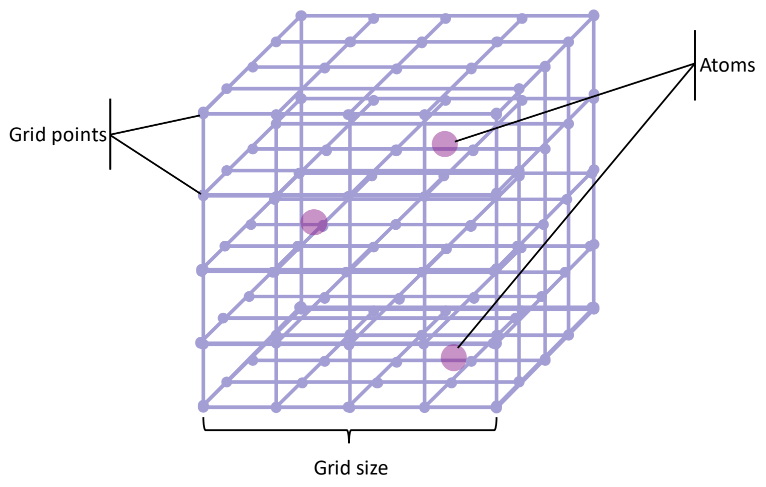


Figure 1 Example 3d grid (4x4x4 cuboid)

Each atom consists of 4 elements: [x, y, z] coordinates and an energy value. The physical effect of an atom on a grid point is calculated by summing up the following values calculated per atom: <energy of atom>/<distance between grid point and atom>.

The application requires an input from the user specifying the grid size; in this case, the shape of the 3d grid is assumed to be a cube, meaning the lengths of all grid edges are identical (e.g., a grid with grid size 4 would be a 4x4x4 cube, as depicted in Figure 1). The user also specifies the number of atoms within the grid.

The application generates the specified number of atoms with randomized values for both their coordinates, and their effect values, then calculates the physical effects of these atoms on all grid points in the given grid, determines the minimal value and returns this minimum along with the grid point with this minimal physical effect.

The application displays the progress of the calculation so that each step in the progress bar is an indication that a grid point was processed.

The application consists of 3 components:

1. A server providing a GUI for progress visualization (e.g., for how many atoms has the effect been calculated so far) and interaction with the user
2. An atom-data generator, which will initialize the app with a set of atoms and their corresponding value tuples
3. A calculator, whose multiple instances calculate the effects for each atom in parallel

The services communicate using Amazon's SQS and SNS. Details on the architectural setup are given in the next section.

2 Application Architecture

In this section, an overview and brief description of the application architecture is given. Please note that this is just an overview showing the main components and their high-level interactions (see Figure 2), and that the parallelization and communication details are omitted.

The application's central (computing) components are three EC2 instances on which the aforementioned three services are running.

The services utilize two AWS messaging services to communicate: SNS and SQS. Both SNS and SQS use the FIFO (first in first out) ordering/processing mechanism, which in this case simply allows the user request that came in first to also be processed first. In essence, the GUI service creates SNS topics which allow "labeling" of data flows based on the user request, and it subscribes an SQS queue to this topic. As portions of the data are processed, intermediate updates from the GridSolver are pushed in form of notifications to the defined topic. In the meantime, the GUI service polls on the subscribed queue for notification messages and displays the progress via progress bar. The specifics of the communication flow and data will later be described in detail.

The GUI service and the AtomGenerator service store relevant data (i.e., grid size, generated atom value tuples and result) to objects in an S3 bucket, which represents the data storage mechanism in this architecture.

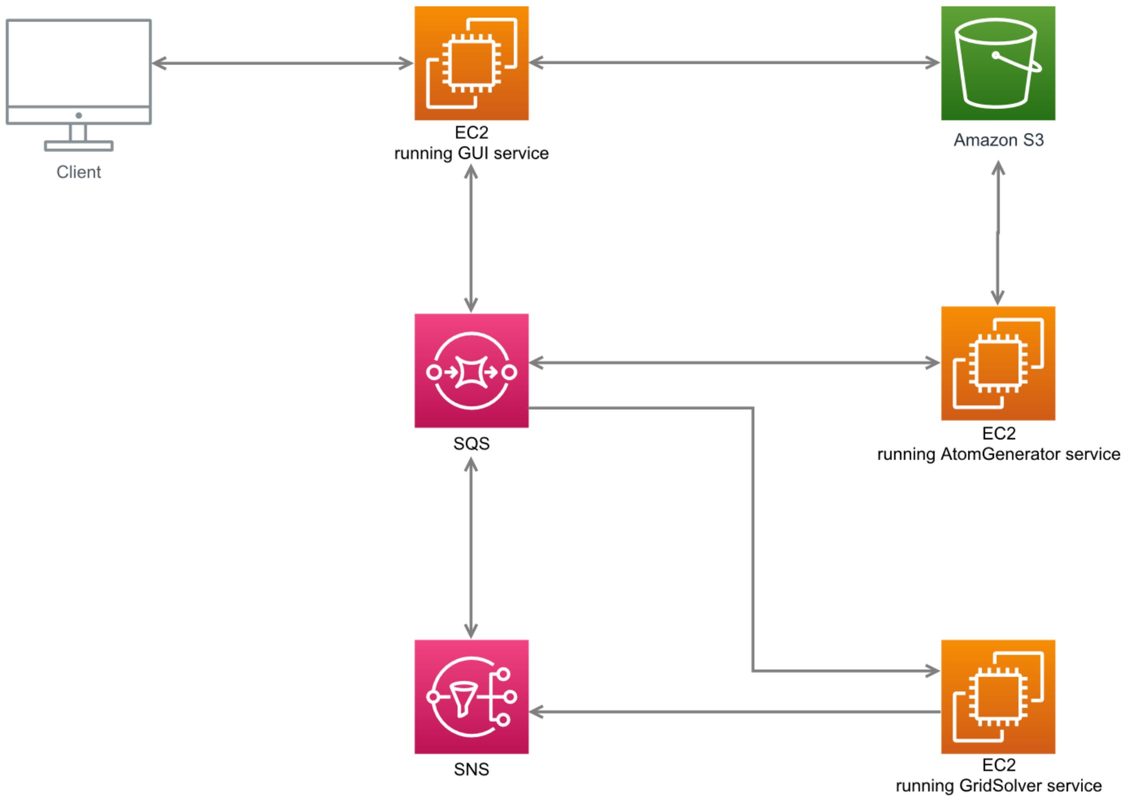


Figure 2 Overall application architecture

2.1 Cloud Design Patterns

The cloud design patterns utilized in this application architecture are

- Pipes-and-Filters, and
- Publish-Subscribe pattern.

2.1.1 Pipes-and-Filters pattern

The usage of this pattern is evident in the integration of Amazon's SQS queues: each queue is a pipe and the services processing the transferred data are the filter components.

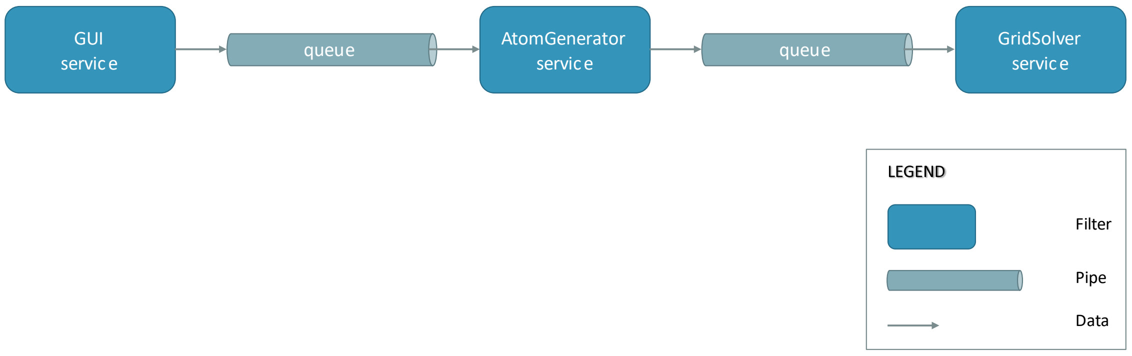


Figure 3 Demonstration of the Pipes-and-Filters pattern usage in the application

2.1.2 Publish-Subscribe pattern

This pattern is applied by using Amazon's SNS (with SQS): the SQS queue from which the GUI service polls for messages is a subscriber to a certain topic, to which the GridSolver service processes send intermediate results in form of "push messages".

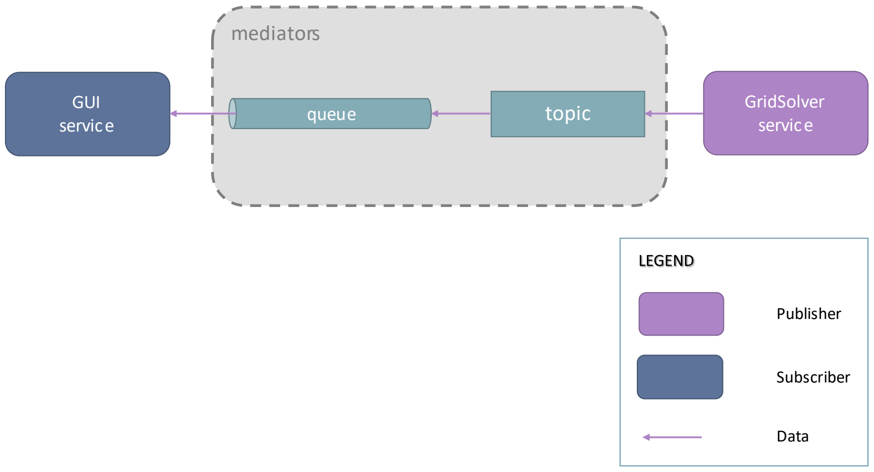


Figure 4 Demonstration of the Publish-Subscribe pattern usage in the application

3 Storage, Queueing and Messaging

This section describes mechanisms used for data storage, queuing and messaging in the application, including elaboration on what exactly is stored and communicated, in which format, and why.

3.1 Data storage

The chosen data storage mechanism is Amazon's S3 which uses *buckets* as storage units and *objects* as the stored data units. This AWS service is simple, comprehensive, highly available, lightweight and fast; which is why it has been chosen for this particular use case, where, for instance, sophisticated security and access mechanisms are not crucial, or even necessary for the application's function.

The data is stored in a separate object for each user request, and is essentially a JSON object which contains the following:

```
{
        'Grid size': <grid size as defined by user>,
        'Atom count': <number of atoms in the grid as defined by user>,
        'Atom values': <array of tuples with atom values, generated by the AtomGenerator>,
        'Result': <calculation result, comprised of grid point coordinates and the min.value>
}
```

3.2 Queueing and Messaging

As mentioned before, the services communicate using SQS and SNS. The communication process is organized as depicted in Figure 5. Note here that the "queue1.fifo" and the "SNS topic" are newly generated for each user, to prevent the messages from different users from interfering with each other and, consequently, breaking the progress visualization per user request.

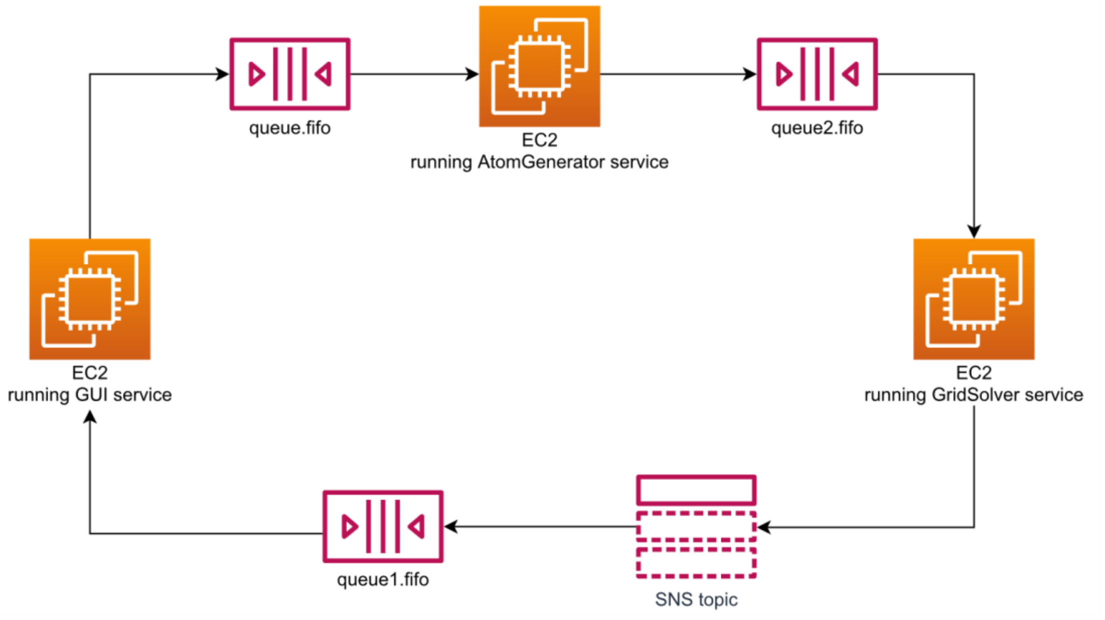


Figure 5 Queuing and messaging among the distributed application components

In particular, the communication process executes in the following manner:

1. GUI service creates an SNS topic named with a UUID
2. GUI service creates a corresponding SQS queue
3. GUI service stores the topic ARN for the current user's session
4. GUI service subscribes the corresponding SQS queue to this topic
5. GUI service creates an object in the predefined S3 bucket with initial data (grid size, atom count)
6. GUI service sends a message containing user-defined data (grid size, atom count) as well as the topic ARN and the object key (ID) in S3, to a queue where the AtomGenerator service is listening.
7. From the given data, the AtomGenerator generates a set of atom tuples {x, y, z, e}
8. AtomGenerator service stores the atom values into the existing object in S3, using the object key.
9. AtomGenerator service forwards the generated atom values, along with the topic ARN and grid size to a queue where the GridSolver is listening.
10. GridSolver service extracts the grid- and atom data from the received message and calculates the result (intra-parallelized)
11. Each subprocess solving a fraction of the problem data sends an SNS update message to the topic using the topic ARN
12. GUI reads from the "subscriber"-queue and updates the progress bar for each update message until all the grid points' updates have been received
13. When all the subprocesses have finished processing the grid data, the global minimum effect and the corresponding grid point are sent as a final push notification to the SNS topic
14. Finally, GUI displays the result read from the final message, and stores the result into the adequate object in S3

4 Parallelization and Cooperation of Application Components

The intra-parallelization is realized by two means within the GridSolver service:

(1) Vectorization (of the atom values):
   a. Performing element-wise operations on 4 vectors containing x, y, z coordinates and energy values
(2) Multiprocessing (of parts of the grid)
   a. Processing a portion of the grid point set per subprocess,

while the inter-parallelization occurs when e.g., the GUI displays the progress page while the AtomGenerator is generating the atom values and forwarding relevant data at the same time, or when the GUI updates the progress bar while the GridSolver processes remaining grid points independently at the same time.

A visualization of the interactions, and (hopefully) a comprehensive view of the parallelly executed aspects is given in Figure 6.

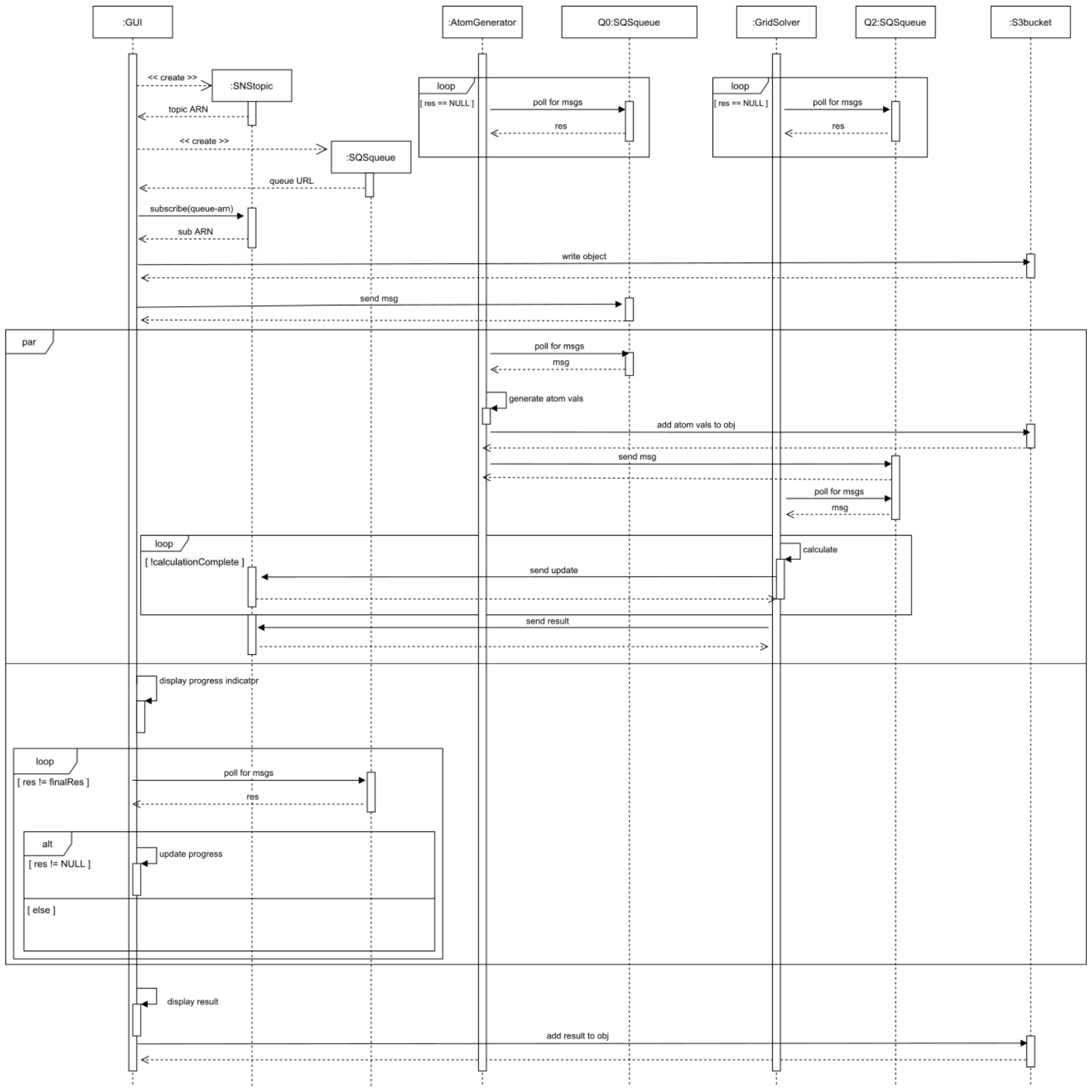


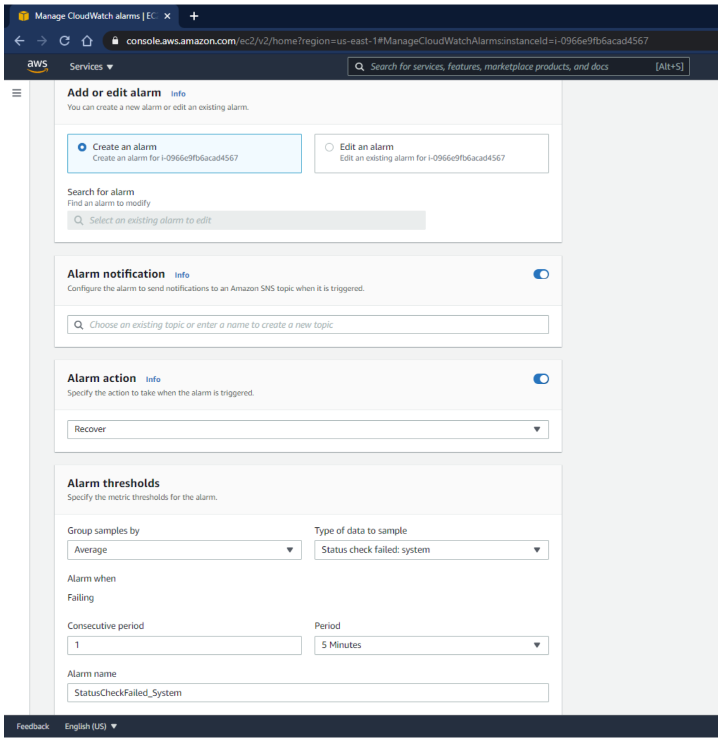
Figure 6 Overview of the communication flow among application services

5 Fault Tolerance

When it comes to fault tolerance, using AWS components such as EC2, SQS, SNS, and S3 in itself already sets grounds for a fault tolerant system. This is due to high availability of the services and provided support for monitoring, load balancing, (auto)scaling according to load/demand for the respective services, and automatic action mechanisms upon crash cases.

In this specific application scenario, the following is applied to ensure fault tolerant functioning:

- For each user request, a separate SQS queue is created that serves as a pipe for update messages and result message for that specific request
- Using FIFO queues and topics ensures the result message is never processed before all intermediate update messages have been processed
- The application services delete the messages only after they have been processed
- Setup of automatic recovery by utilizing CloudWatch alarms enables the EC2 instances to be recovered upon system failure (see Figure 7)


Manage CloudWatch alarms | EC
console.aws.amazon.com/ec2/v2/home?region=us-east-1#ManageCloudWatchAlarms:instanceId=i-0966e9fb6acad4567
aws
Services
Search for services, features, marketplace products, and docs
[Alt+S]

Add or edit alarm Info
You can create a new alarm or edit an existing alarm.

[✓ Create an alarm] Create an alarm for i-0966e9fb6acad4567
Edit an alarm — Edit an existing alarm for i-0966e9fb6acad4567

Search for alarm
Find an alarm to modify
Select an existing alarm to edit

Alarm notification Info
Configure the alarm to send notifications to an Amazon SNS topic when it is triggered.
Choose an existing topic or enter a name to create a new topic

Alarm action Info
Specify the action to take when the alarm is triggered.
Recover

Alarm thresholds
Specify the metric thresholds for the alarm.

Group samples by: Average
Type of data to sample: Status check failed: system
Alarm when: Failing
Consecutive period: 1
Period: 5 Minutes
Alarm name: StatusCheckFailed_System

Figure 7 Setting a CloudWatch alarm for system failure

- Additionally, the applications on the EC2 instances are configured as *ubuntu services* and are automatically started (without manual intervention/starting the apps via ssh) at boot time.
  - This was set up by passing bash commands to the “user data” of the instances. For example, in the case of the GUI service, the following user data was defined:

    ```
    #cloud-boothook
    #!/bin/bash
    service gui restart
    ```
  - This ensures that even after a crash, and subsequent reboot during recovery, the services will be available again.

6 Test Cases, Results and Resource Utilization

Some of the test data and results for the application are shown in Table 1.

Table 1 Test data and results

Grid size	Total number of grid points	Atom count[1]	Time/request processed[2][seconds]
4	64	20, 50, 100, 500, 1000	~4
8	512	20, 50, 100, 500, 1000	~23
12	1728	20, 50, 100, 500, 1000	~67
16	4096	20, 50, 100, 500, 1000	~150

Utilization of resources (in particular, CPU usage and network i/o traffic), can be seen in Figures 8 to 10.

What can be easily seen in the metrics visualizations is that the AtomGenerator's CPU resource is more utilized than that of the other instances, which is probably due to the fact that its instance type is t2.micro which has only 1 vCPU, while the other 2 instances are t3.micro instances with 2 vCPUs. The GridSolver, performing compute-intensive (but parallelized and optimized) operations is the second instance in CPU utilization, while the GUI utilizes the least CPU power for its function.

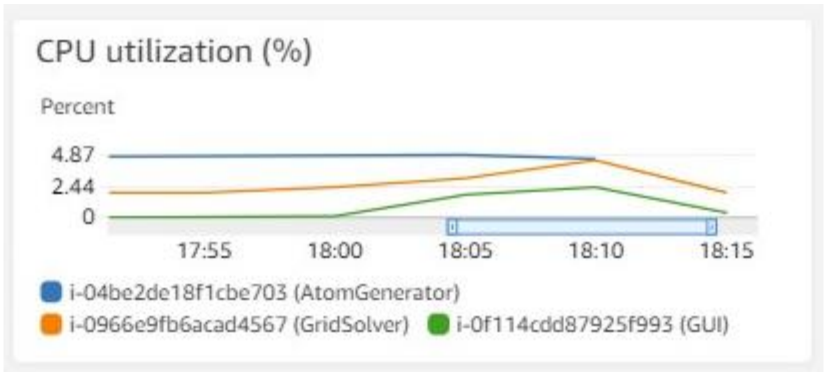


Figure 8 CPU utilization in % for each of the EC2 instances

When it comes to network traffic, one can observe that the GUI mostly receives packets (this is a result of reading update and result data from the SQS queue), while the GridSolver sends the most packets (because is pushes intermediate updates, as well as the final result to the respective topic per request).

Note that AtomGenerator's and GridSolver's metrics values are continuously above 0 for each resource, because the constantly query/listen at their respective SQS queues for incoming requests, while the GUI only starts listening for messages after the user has triggered the actual calculation of the result.

[1] Thanks to the utilized vectorization of the atom values, the operations on greater number of atoms does not (significantly) affect the processing time
[2] Note that this refers to the progress bar update speed, i.e. elapsed time from 0 to max. progress indicator value

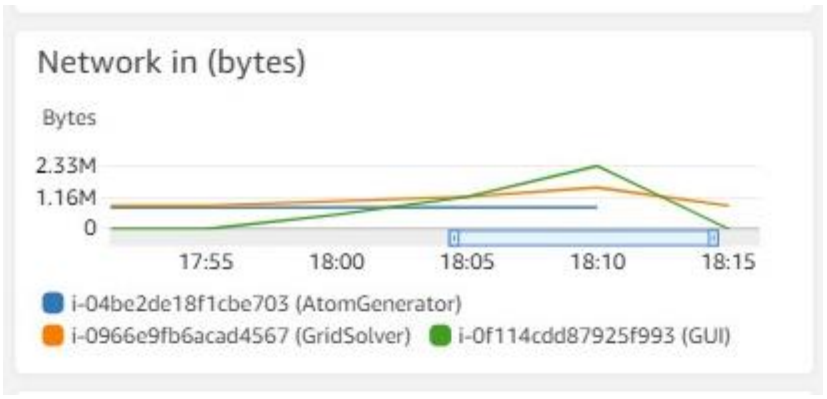


Figure 9 Inbound network traffic in bytes for each of the EC2 instances

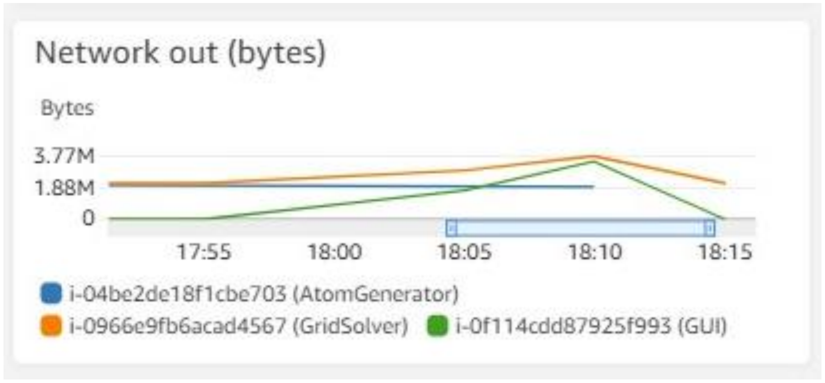


Figure 10 Outbound network traffic in bytes for each of the EC2 instances

7 Deployment

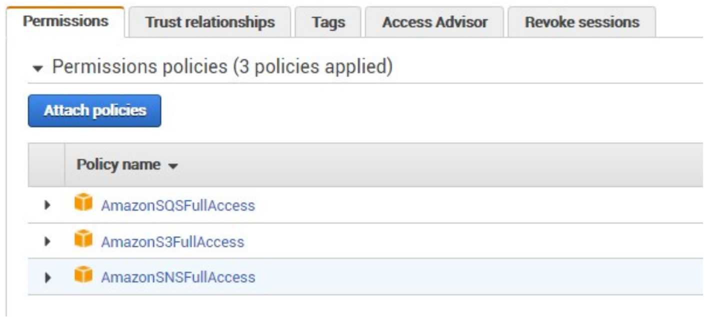
7.1 Background: Used technologies

- All 3 services are written in Python
- The GUI service additionally utilizes the Flask web framework
- The boto3[3] AWS SDK for Python is used in all 3 services to make use of AWS services
- Other relevant packages are numpy and multiprocessing

7.2 Deploying to AWS

The overall deployment process was comprised roughly of the following steps (note that here the intermediate troubleshooting and adaptation steps are omitted, and will be explained in the next section):

1. Create IAM role for EC2 instances (this enables them to use/access AWS APIs without manually updating and hardcoding credentials and access keys)
    - The permissions included are full accesses to SQS, S3 and SNS services:



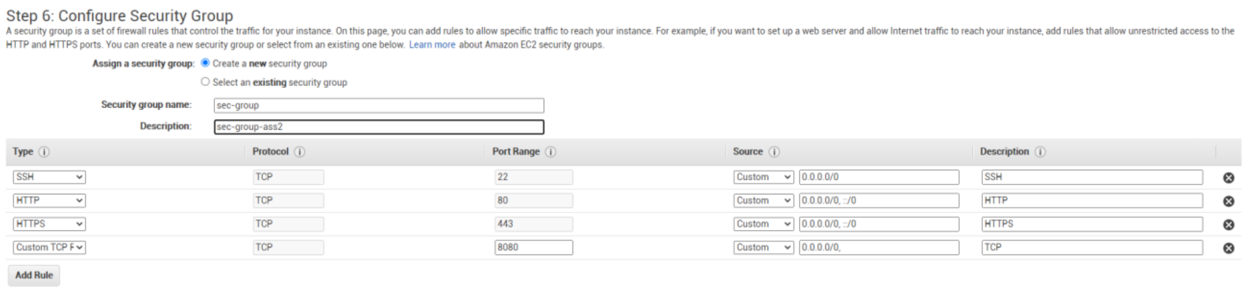
Permissions | Trust relationships | Tags | Access Advisor | Revoke sessions

Permissions policies (3 policies applied)

Attach policies

Policy name
AmazonSQSFullAccess
AmazonS3FullAccess
AmazonSNSFullAccess

2. Launch EC2 instances
    - All instances use the Ubuntu Server 20.04 LTS AMI
    - The GUI and the GridSolver instances are t3.micro instances, while the AtomGenerator instance is a t2.micro instance. This choice was based on (1) number of vCPUs and (2) network performance, although for this application it does not make a big difference in overall performance.
    - Assign the IAM role to each of the instances
    - Storage with default settings
    - Security Group configuration is set up for the GUI instance, since it should be accessible from the internet:



Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. Learn more about Amazon EC2 security groups.

Assign a security group: [✓ Create a **new** security group] Select an **existing** security group

Security group name: sec-group

Description: sec-group-ass2

Type	Protocol	Port Range	Source		Description
SSH	TCP	22	Custom	0.0.0.0/0	SSH
HTTP	TCP	80	Custom	0.0.0.0/0, ::/0	HTTP
HTTPS	TCP	443	Custom	0.0.0.0/0, ::/0	HTTPS
Custom TCP F	TCP	8080	Custom	0.0.0.0/0,	TCP

Add Rule

    - Generate key pair for SSH access
3. Create Elastic IPs and assign them to the EC2 instances

[3] https://aws.amazon.com/sdk-for-python/?nc1=h_ls

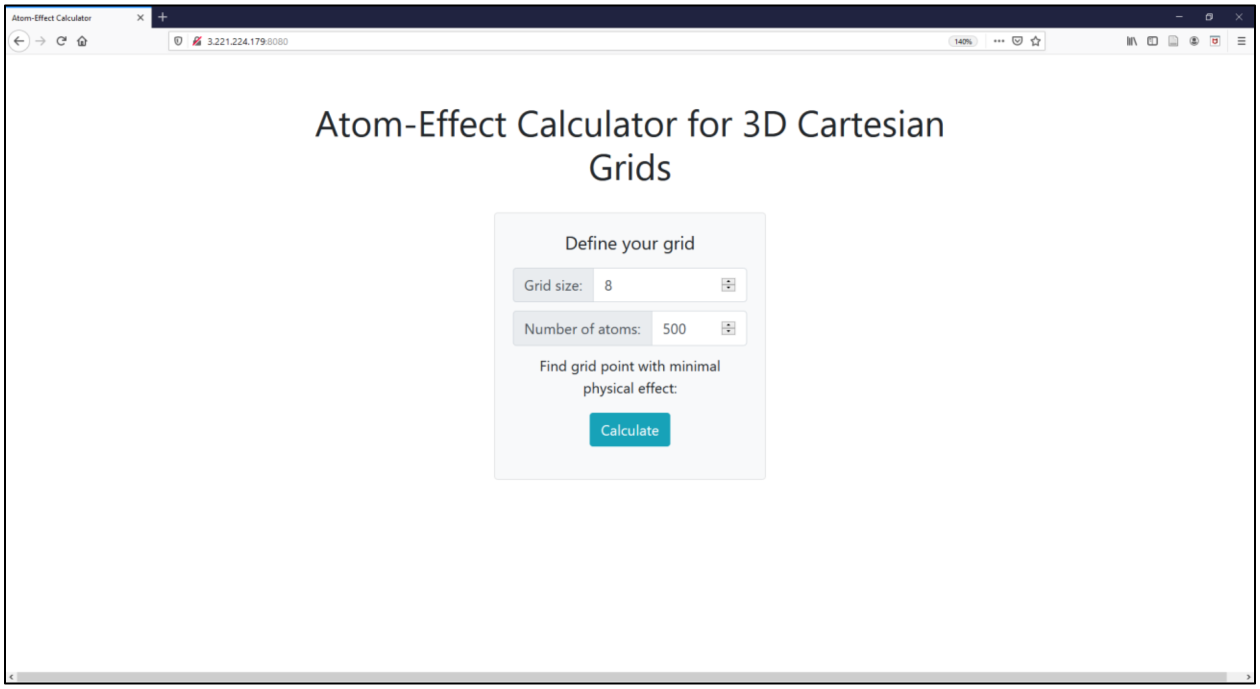
4. Start instances
5. Connect to the instances via SSH and set up the environment for app deployment
    - Run apt update
    - Install python3 and pip
    - Create dedicated directories for application code
    - Transfer application data from local computer (for this, sftp via FileZilla[4] was used)
    - Initialize virtual environments for the apps and install required packages using pip
    - Create ubuntu services from the python code to be able to manage them via
      ```
      service <app_name> start
      service <app_name> stop
      service <app_name> restart
      service <app_name> status
      ```
6. Stop the instances
7. Edit user data
    - Insert the following to start the services at boot up of the EC2 instances:
      ```
      #cloud-boothook
      #!/bin/bash
      service <app_name> restart
      ```
8. Configure CloudWatch alarms to automatically recover instances upon failure
9. Start the instances

7.3 The End Product

Finally, to describe the "look-and-feel" of the deployed system, Figures 11 to 15 present screenshots of the application "in action".

When running, the application is available at http://3.221.224.179:8080.

[4] https://filezilla-project.org/



Atom-Effect Calculator for 3D Cartesian Grids

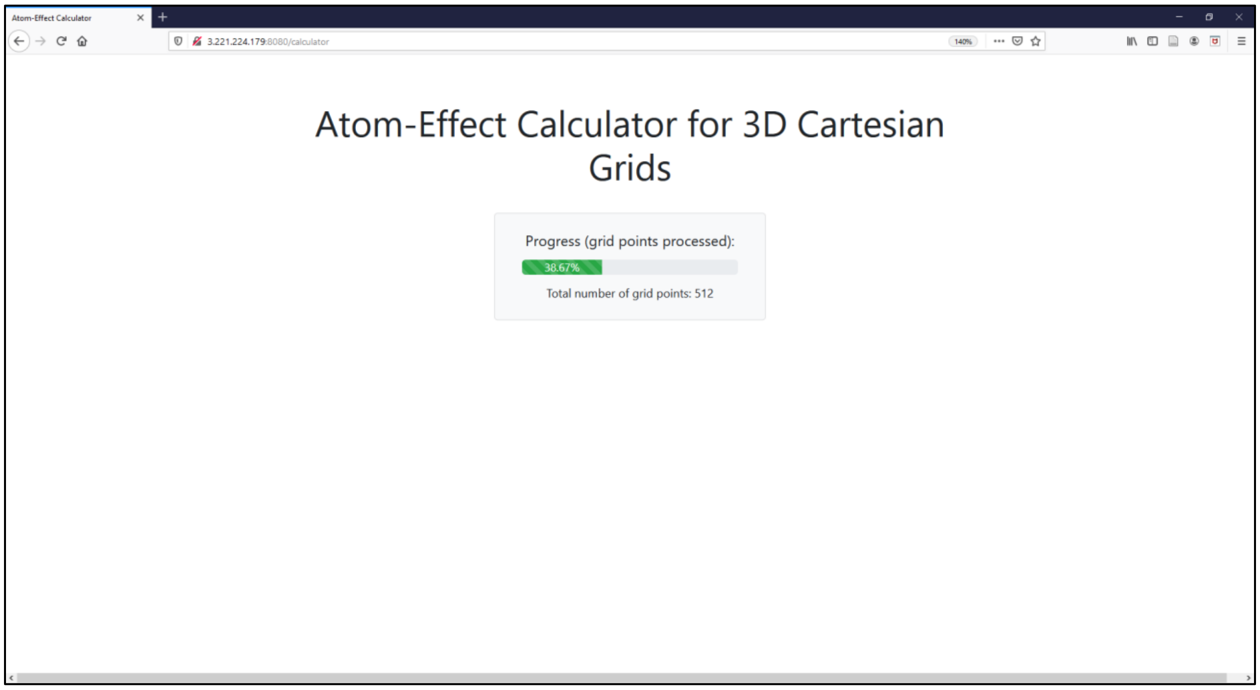
Define your grid

Grid size: 8

Number of atoms: 500

Find grid point with minimal physical effect:

Figure 11 Main page of the application



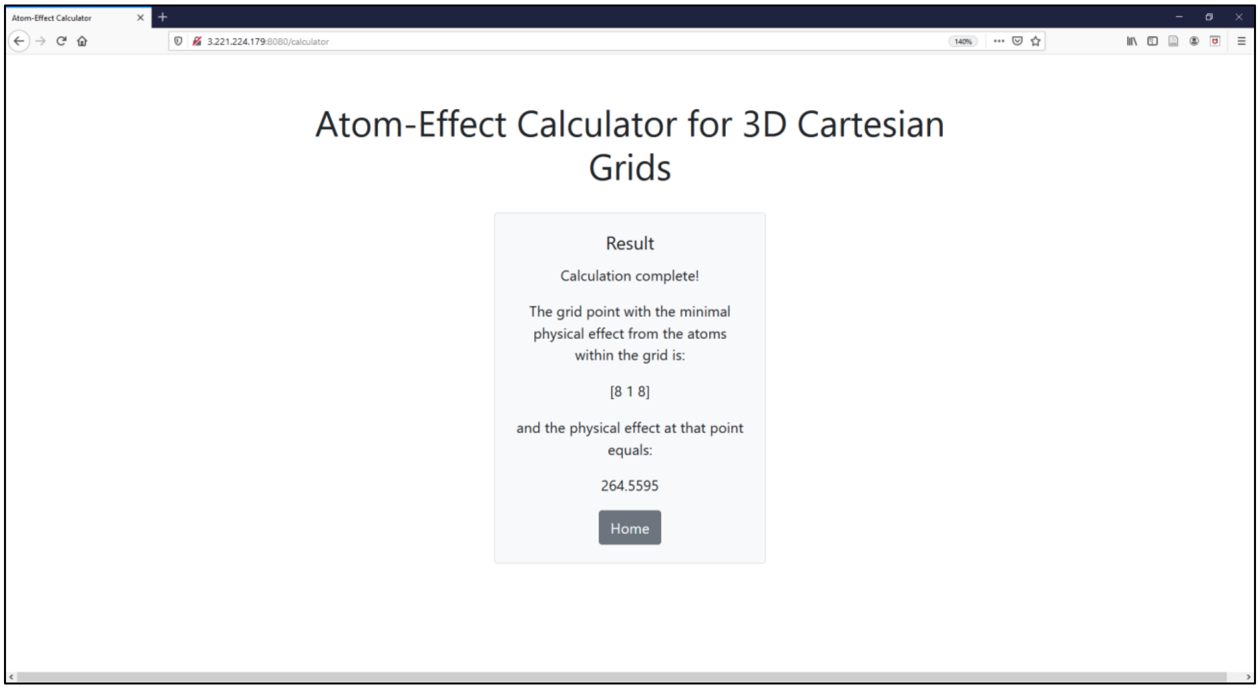
Atom-Effect Calculator for 3D Cartesian Grids

Progress (grid points processed):

38.67%

Total number of grid points: 512

Figure 12 Progress indicator page



Atom-Effect Calculator for 3D Cartesian Grids

Result

Calculation complete!

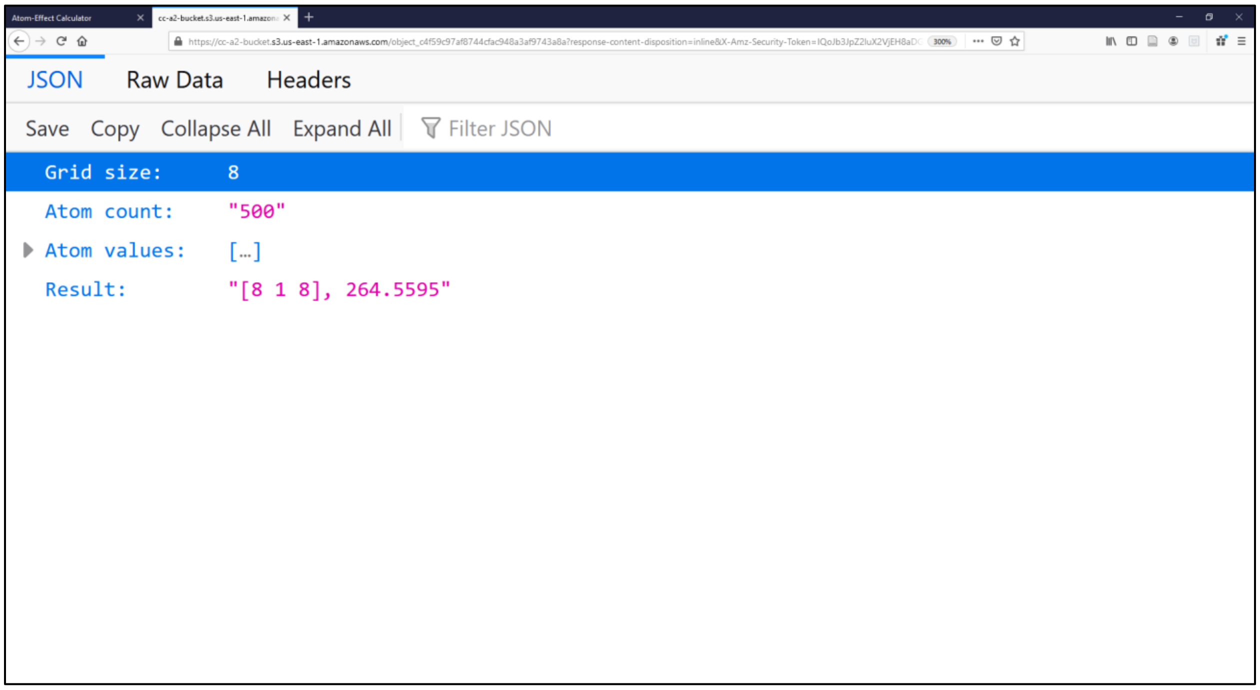
The grid point with the minimal physical effect from the atoms within the grid is:

[8 1 8]

and the physical effect at that point equals:

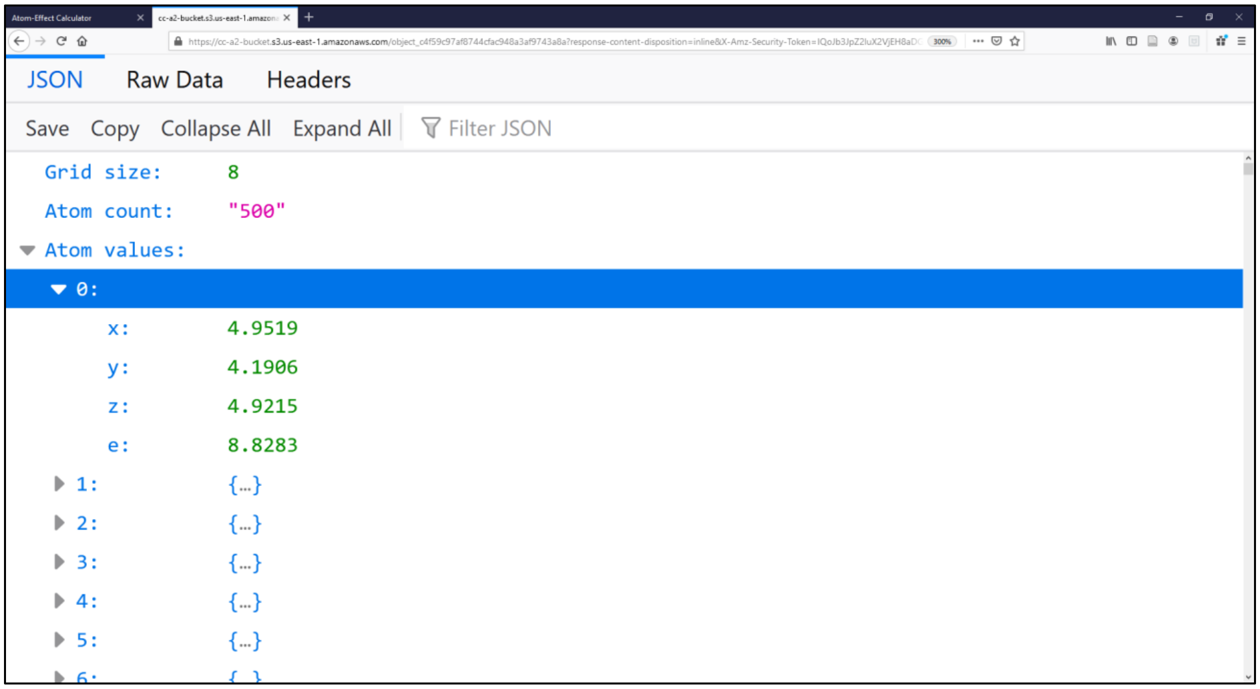
264.5595

Figure 13 Result page



```
Grid size:      8
Atom count:     "500"
Atom values:    […]
Result:         "[8 1 8], 264.5595"
```

Figure 14 All relevant data stored in a new object in the S3 bucket



```
Grid size:      8
Atom count:     "500"
▼ Atom values:
  ▼ 0:
      x:        4.9519
      y:        4.1906
      z:        4.9215
      e:        8.8283
  ▶ 1:          {…}
  ▶ 2:          {…}
  ▶ 3:          {…}
  ▶ 4:          {…}
  ▶ 5:          {…}
```

Figure 15 The new object with an expanded atom value item

8 Troubleshooting: Key Issues and Solutions

The key issues that arose during development and subsequent deployment, as well as their solutions, are briefly described in Table 2.

Table 2 Key issues and solutions

#	ISSUE	PROBLEM DESCRIPTION	SOLUTION
1	Progress update consistency per user	At first, only one queue was created that was used by the GUI service to query messages. During initial testing with multiple user requests, it became clear that this is an inadequate approach since each request should have its own, dedicated stream of data to enable consistency in progress visualization and also the final result.	Create one topic and corresponding queue per user request.
2	ElasticBeanstalk deployment	The first deployment attempt was conducted using ElasticBeanstalk (via AWS console, where the existing app can be uploaded as a .zip bundle). However, it appears that this "type" of deployment (i.e., uploading an *existing* application) requires extra configuration files which are not well documented.	Instead of using ElastingBeanstalk, move to "traditional" EC2 launching and setup the instances manually via SSH.
3	Non-static IPs	Each time the EC2 instance is restarted, it gets assigned a new public IP address.	Use Elastic IPs
4	SSE with Flask in production environment	For realizing the progress bar visualization in the GUI, server-sent events (SSE) are used (just Python in the main app and JS in the html, not the Flask SSE library). The problem arose when running the app in a production setup using nginx and gunicorn: the progress bar was not continuously updated or not at all (for grid size = 4), which defeats the purpose of showing the progress bar in the first place.	The production environment setup requires more configuration[5] and code refactoring than was feasible, so instead of this, the built-in Flask server was used and served at port 8080 (which was enabled via security group settings in the beginning).
5	Automatic service launching on the EC2 instances	Starting an application on an EC2 instance is as simple as it is on a local computer. However, after closing the SSH session, the application stops executing. This poses an availability, as well as a fault tolerance issue. The goal is to	Create services from existing apps, by defining `.service` files in `/etc/system/system/` Pass the bash command for restarting the services to the user data of the EC2 instances and include the boot-hook

[5] The SSE Flask package has to be utilized to work in a production environment using ngnix and gunicorn, so the code on the EC2 instance has to be changed. For local development, the built-in Flask server will not work with SSE. https://readthedocs.org/projects/flask-sse/downloads/pdf/latest/
Additionally, the Celery library would have to be utilized for proper visualization of the progress bar.

		start the apps automatically on every boot of the EC2 instance and keep them running.	keyword to make the command run on each boot.