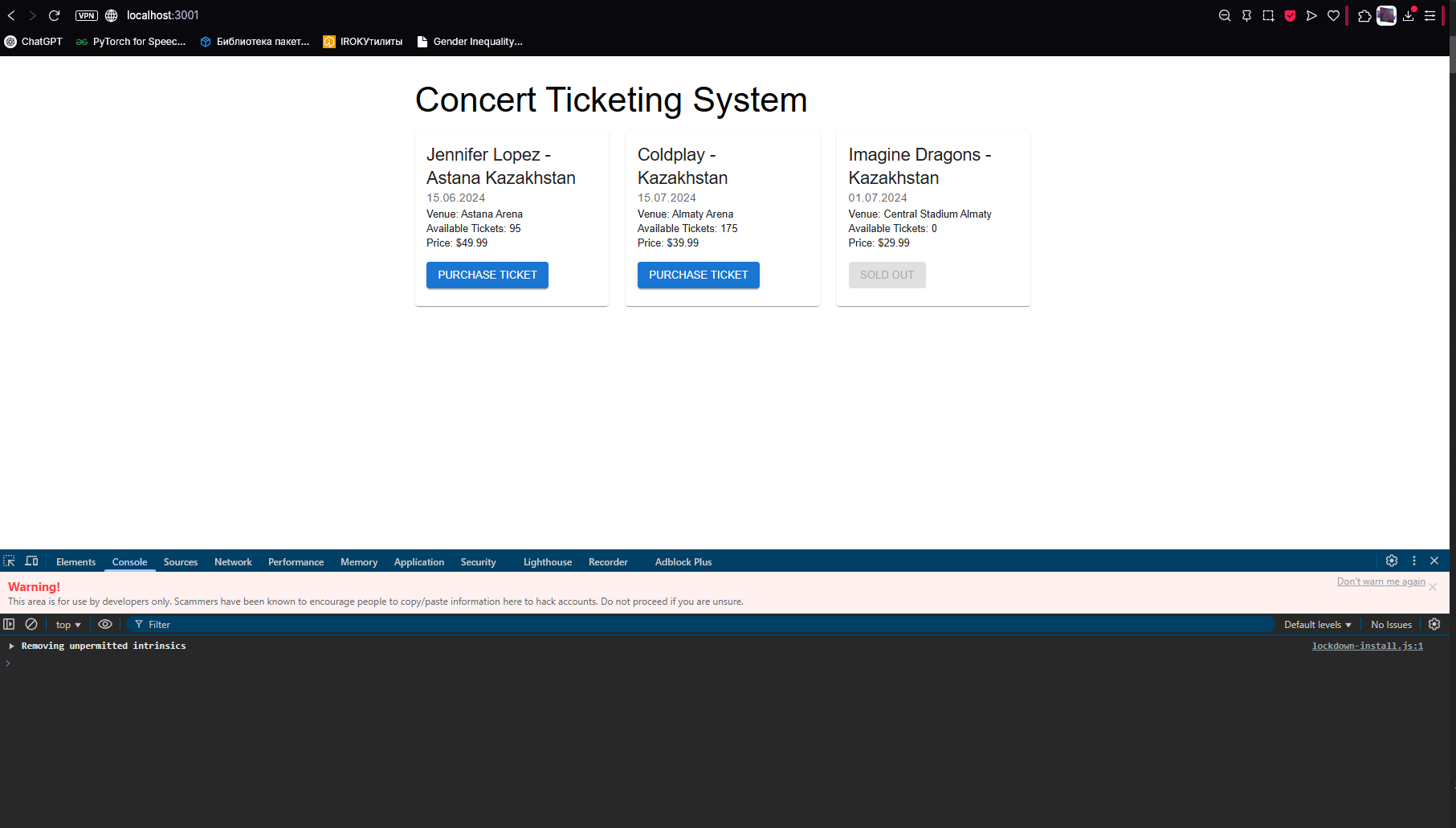
SREFINALZHAS:

TASK 1

1) created a concert ticketing web application

Components: React frontend, FastAPI backend, PostgreSQL database (RDS), Containerized with Docker, Deployed on AWS ECS Fargate



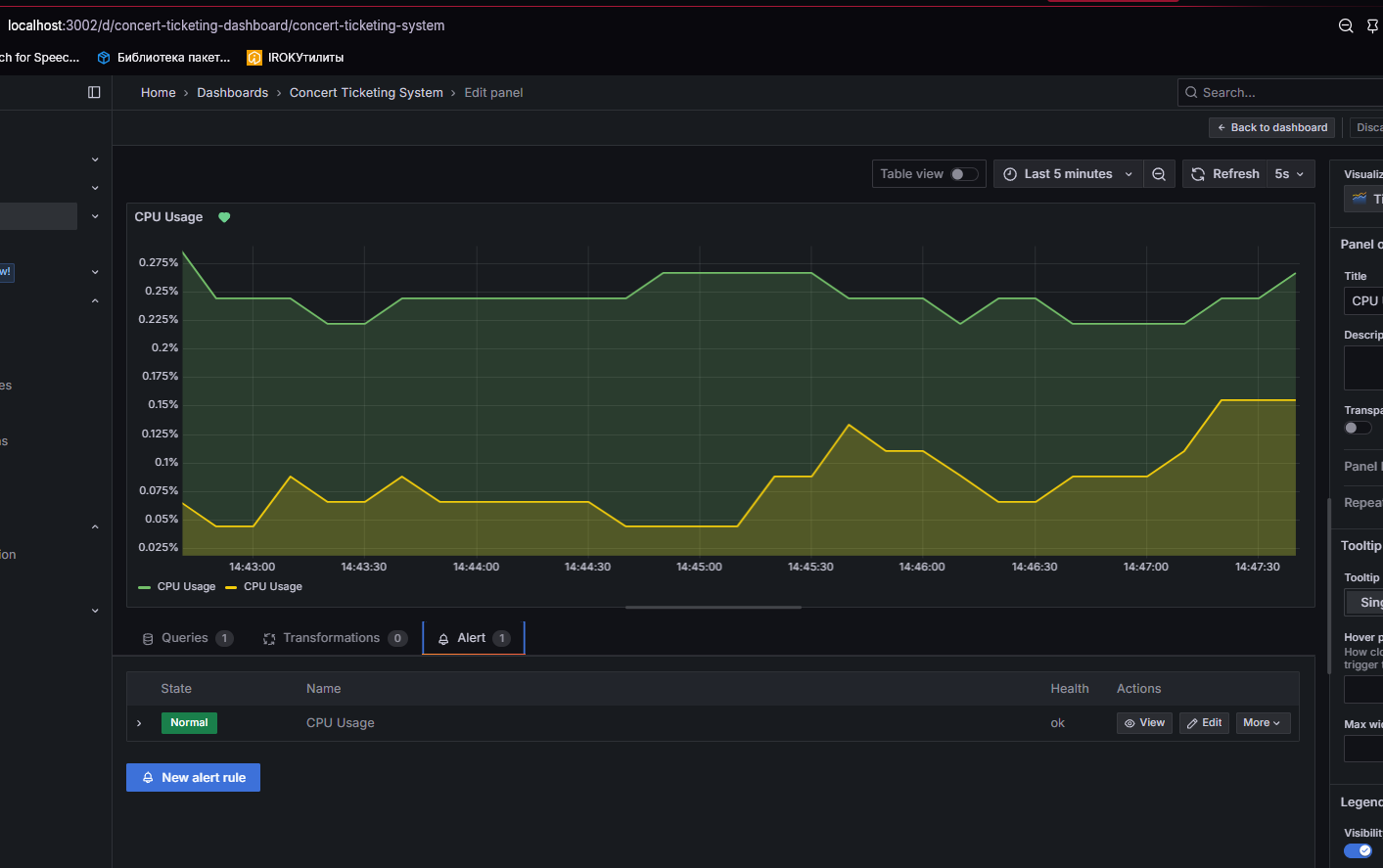
2) Monitoring & Alerting:

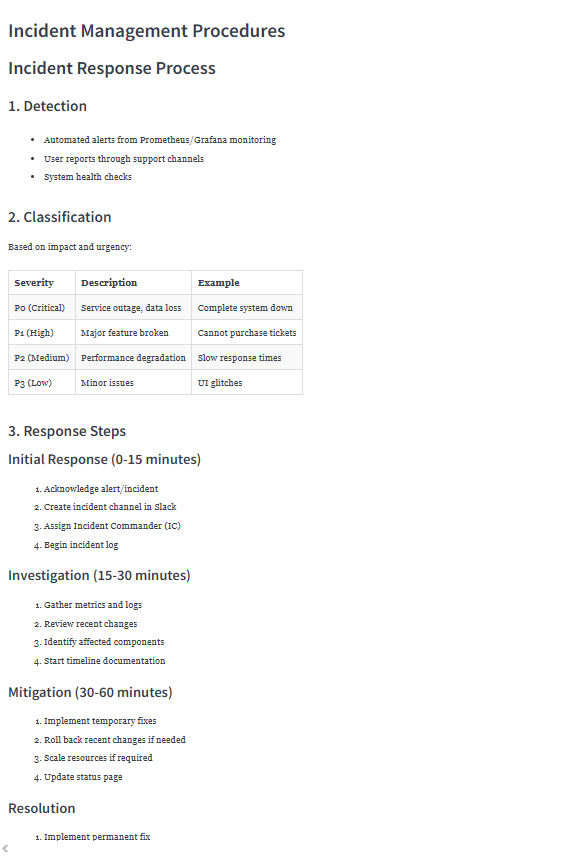
Comprehensive SRE Documentation, Detailed SLI/SLO definitions, SLA with service commitments, Incident management procedures, Postmortem template, Error budget policies

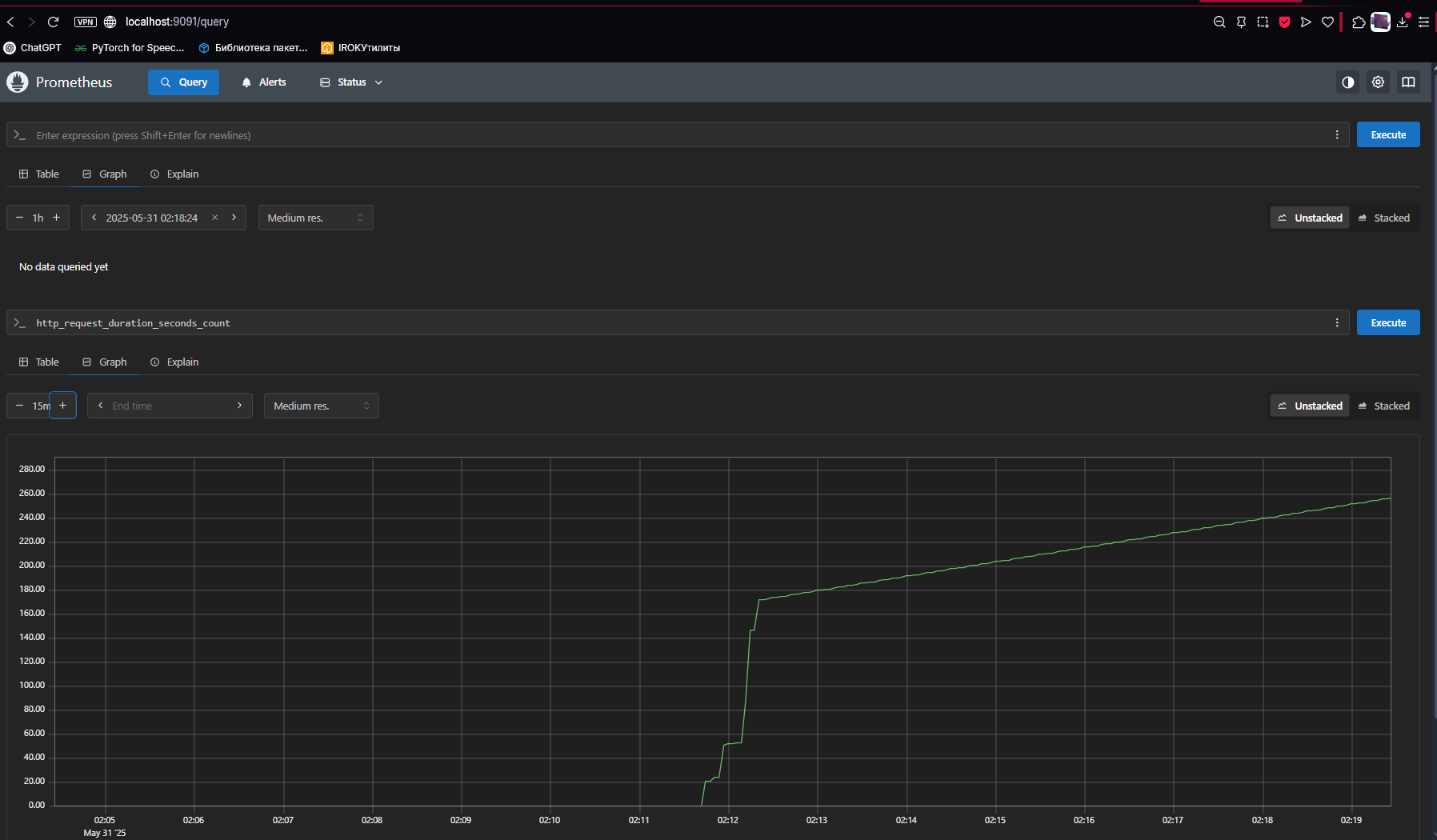
Created post-mortem-incident.md with a simulated incident.

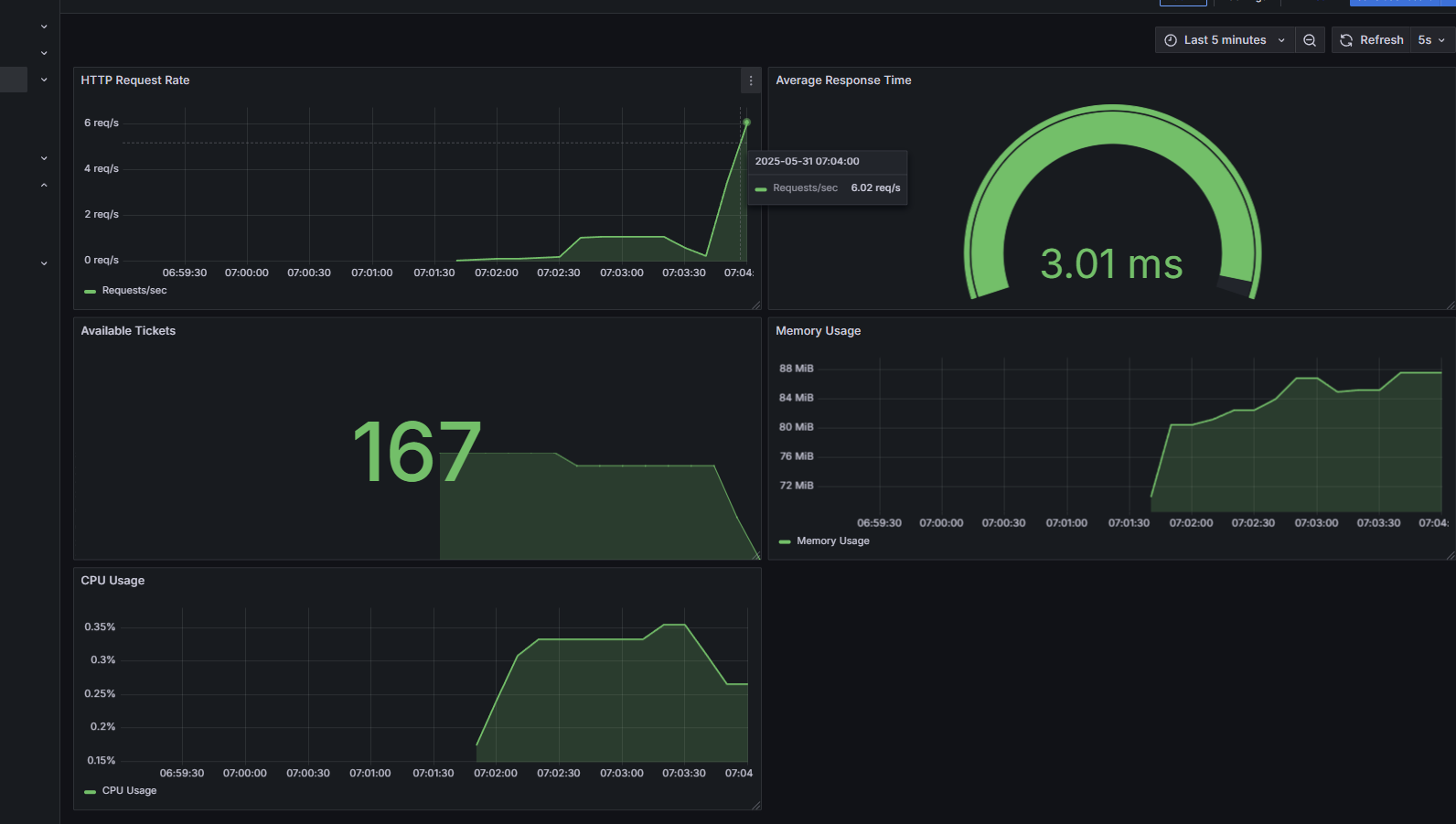
Alerting:











Defined key metrics:

HTTP Request Rate

Average Response Time

Available Tickets

CPU Usage

Memory Usage

**Task 2 - Infrastructure as Code Collaboration**

Application & Infrastructure Design:

Multi-container architecture using Docker Compose

Services: Frontend, Backend, Database, Prometheus, Grafana

High availability considerations through container orchestration

IaC Implementation:

Used Docker Compose for infrastructure definition

Services are containerized and easily deployable

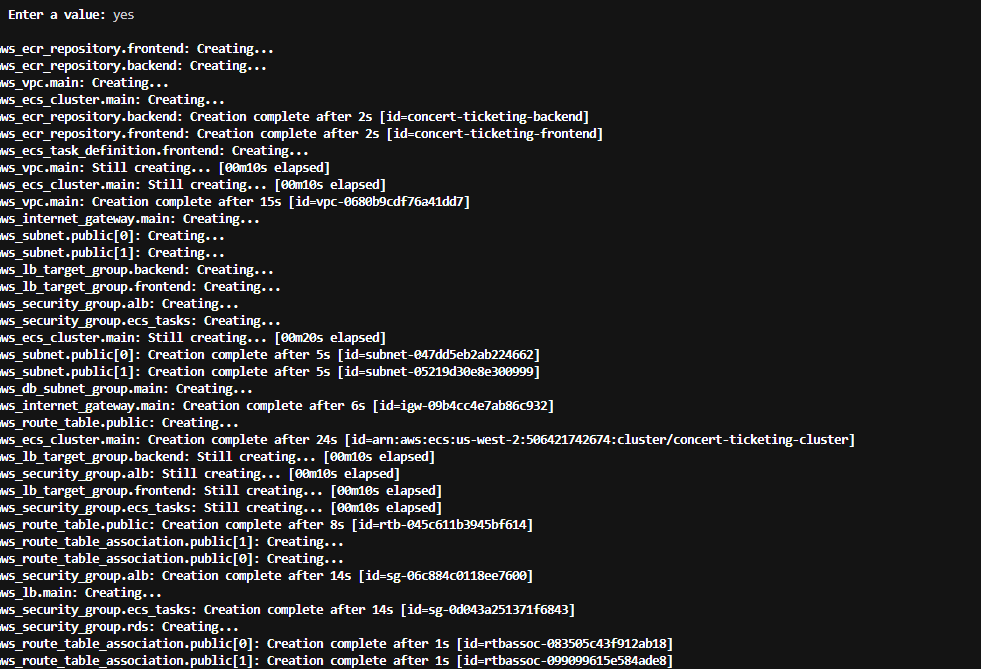
**AWS Infrastructure (Terraform):**

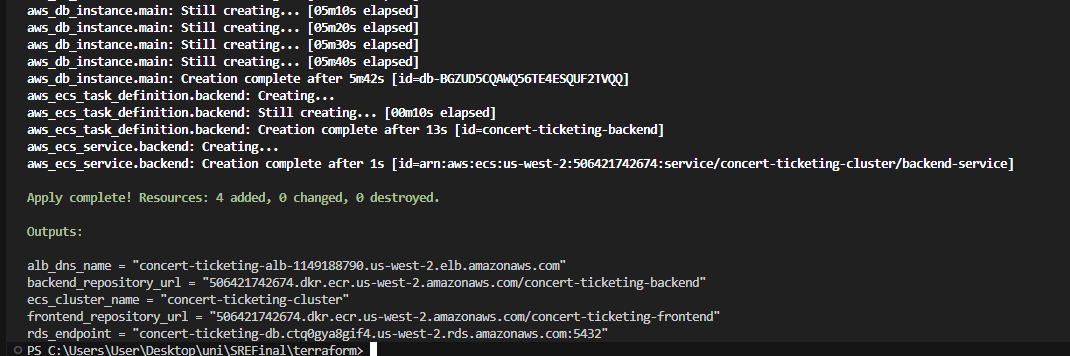
* VPC with public subnets across multiple AZs
* ECS Fargate cluster for container orchestration
* ECR repositories for Docker images
* RDS PostgreSQL database
* Application Load Balancer
* Security groups and networking

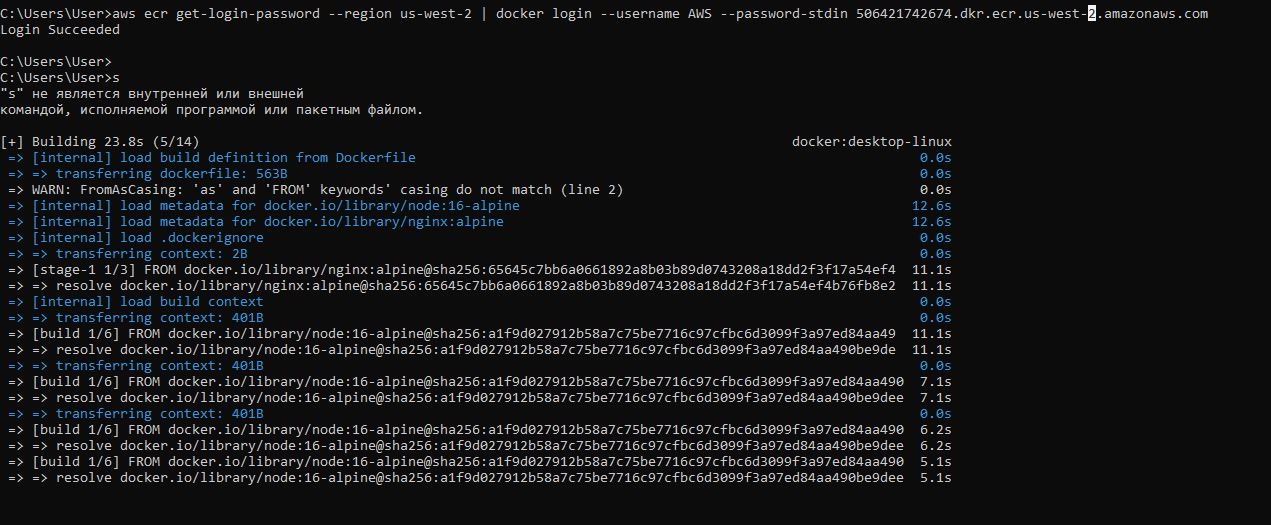
Security Considerations:

* Proper security groups configured
* IAM roles and policies implemented
* Network isolation with VPC
* Secure database access

Terraform log:







Outputs:

alb\_dns\_name = "concert-ticketing-alb-1149188790.us-west-2.elb.amazonaws.com"

backend\_repository\_url = "506421742674.dkr.ecr.us-west-2.amazonaws.com/concert-ticketing-backend"

ecs\_cluster\_name = "concert-ticketing-cluster"

frontend\_repository\_url = "506421742674.dkr.ecr.us-west-2.amazonaws.com/concert-ticketing-frontend"

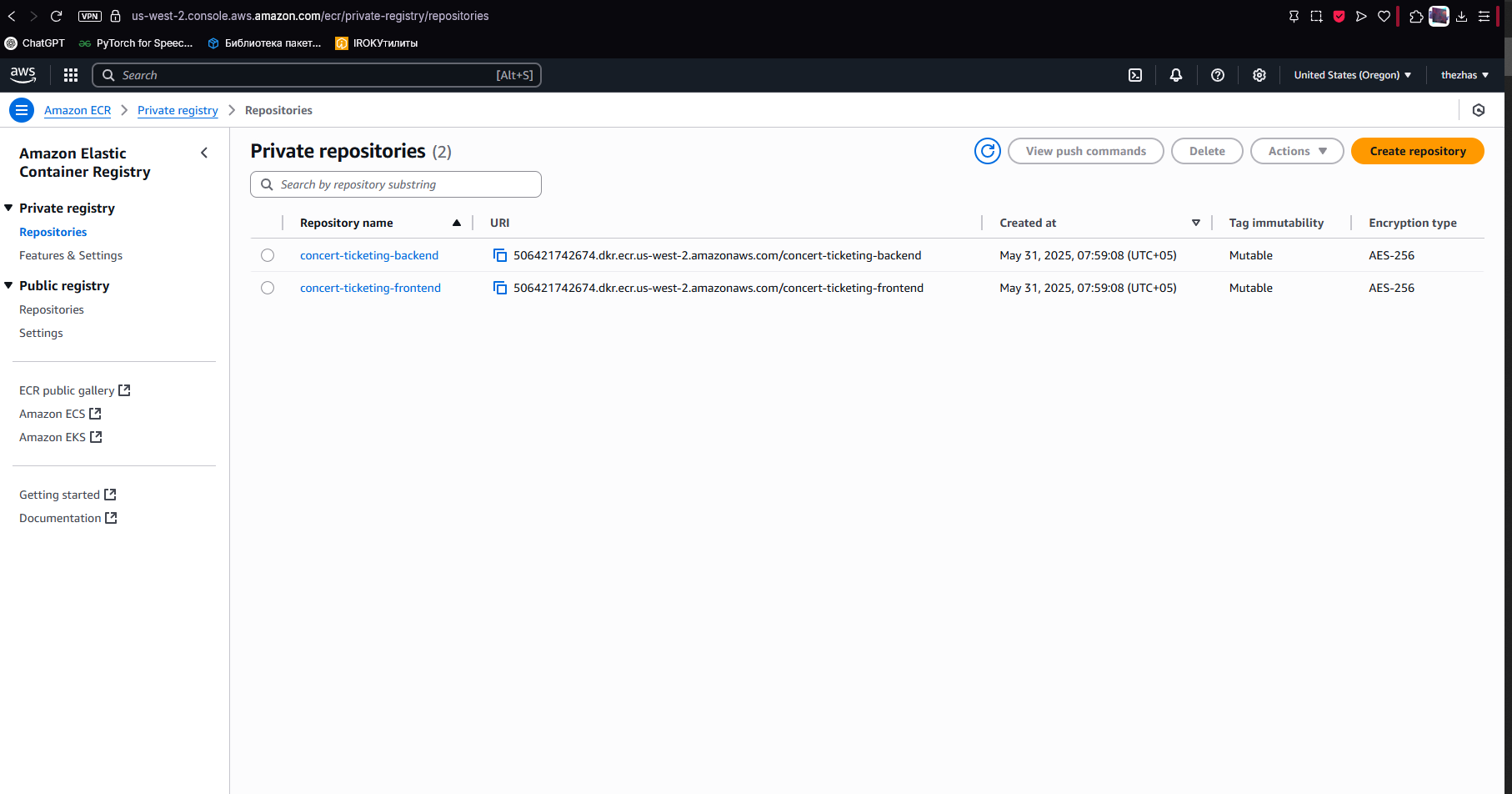
rds\_endpoint = "concert-ticketing-db.ctq0gya8gif4.us-west-2.rds.amazonaws.com:5432"

Integration & Deployment:

All services are integrated and work together

Environment variables and networking properly configured

Volumes for data persistence

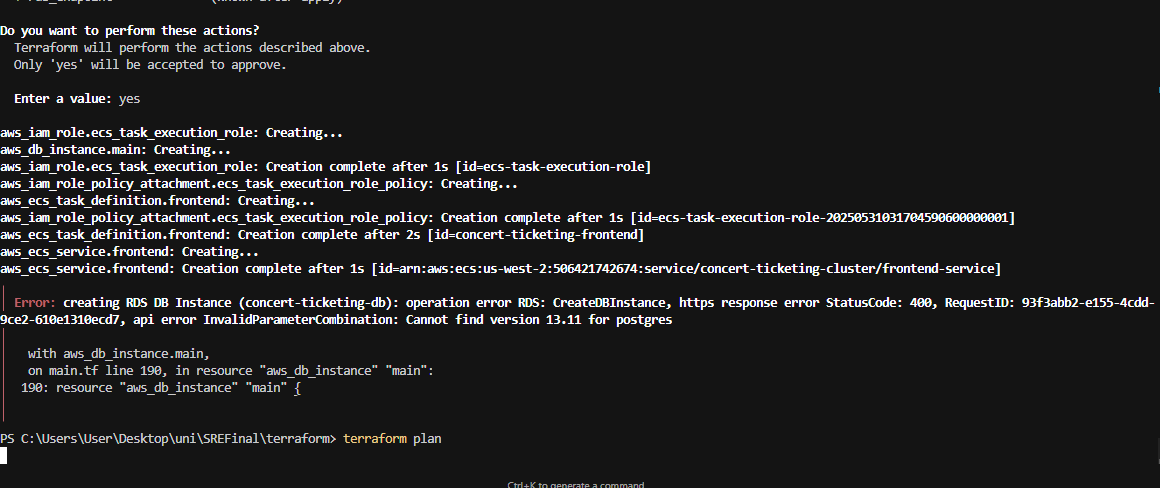


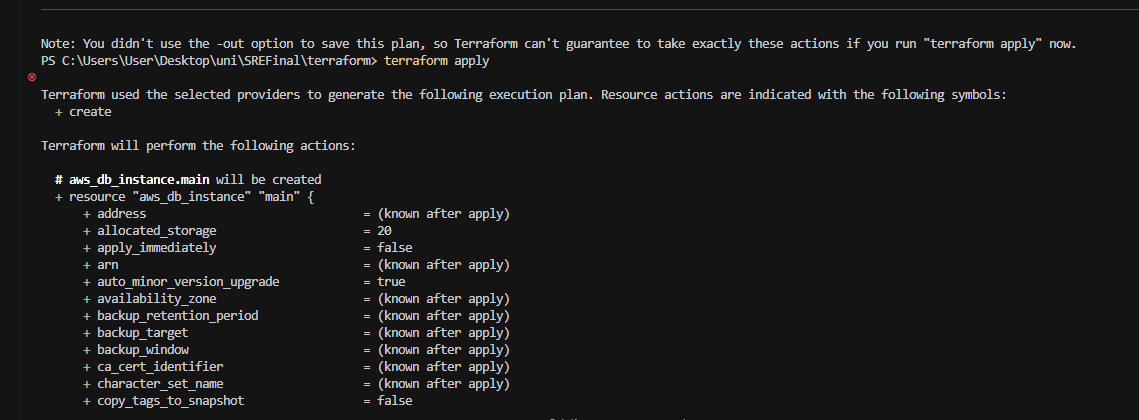
Documentation:

Used best practices

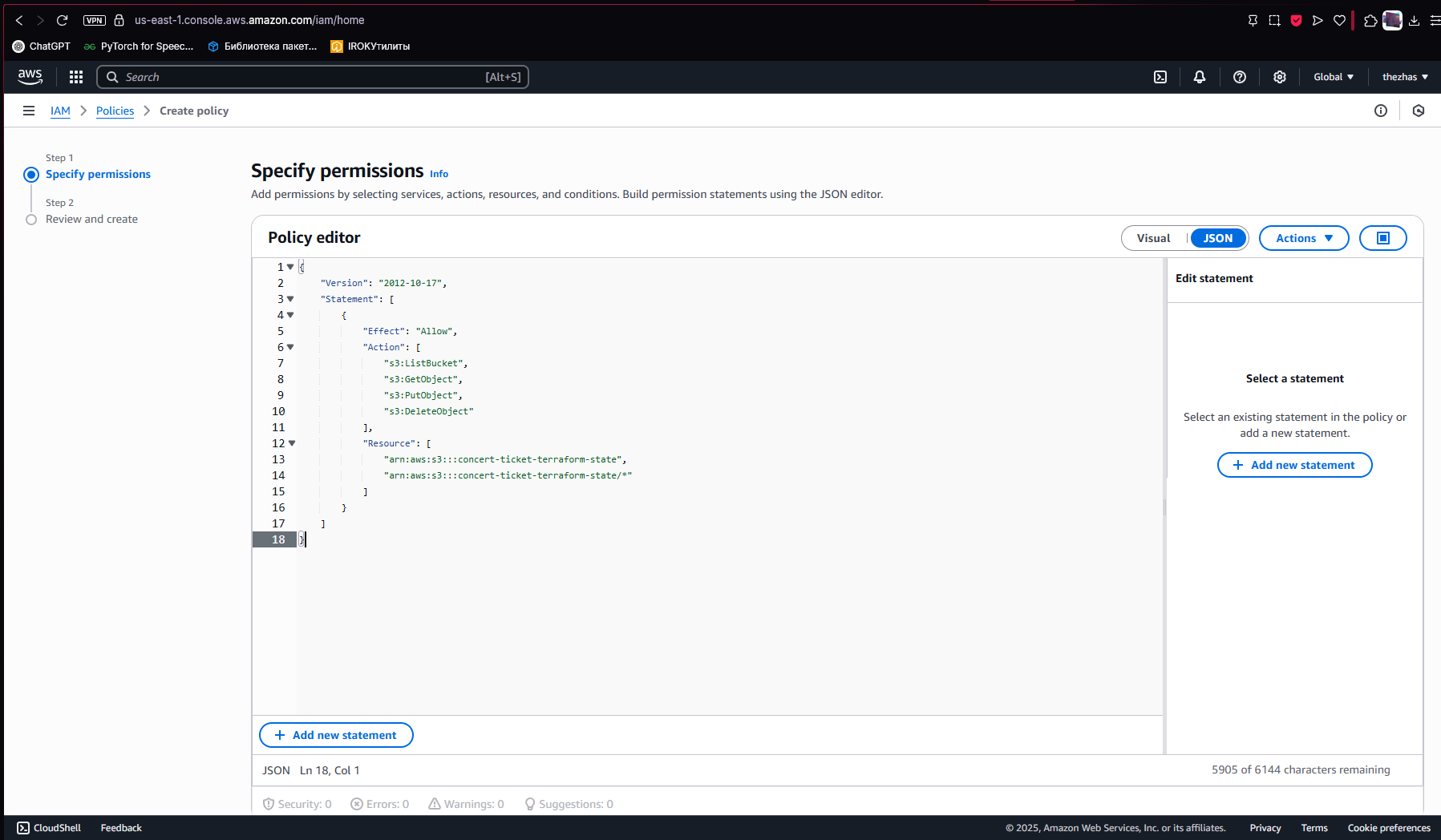
Problems and challenges faced: CORS problems between frontend and backend. Fixed it creating provider for Grafana.  
  
Counter dec() function for tickets error. Fixed this by using a Gauge instead of a Counter for the available tickets metric since it can both increase and decrease:

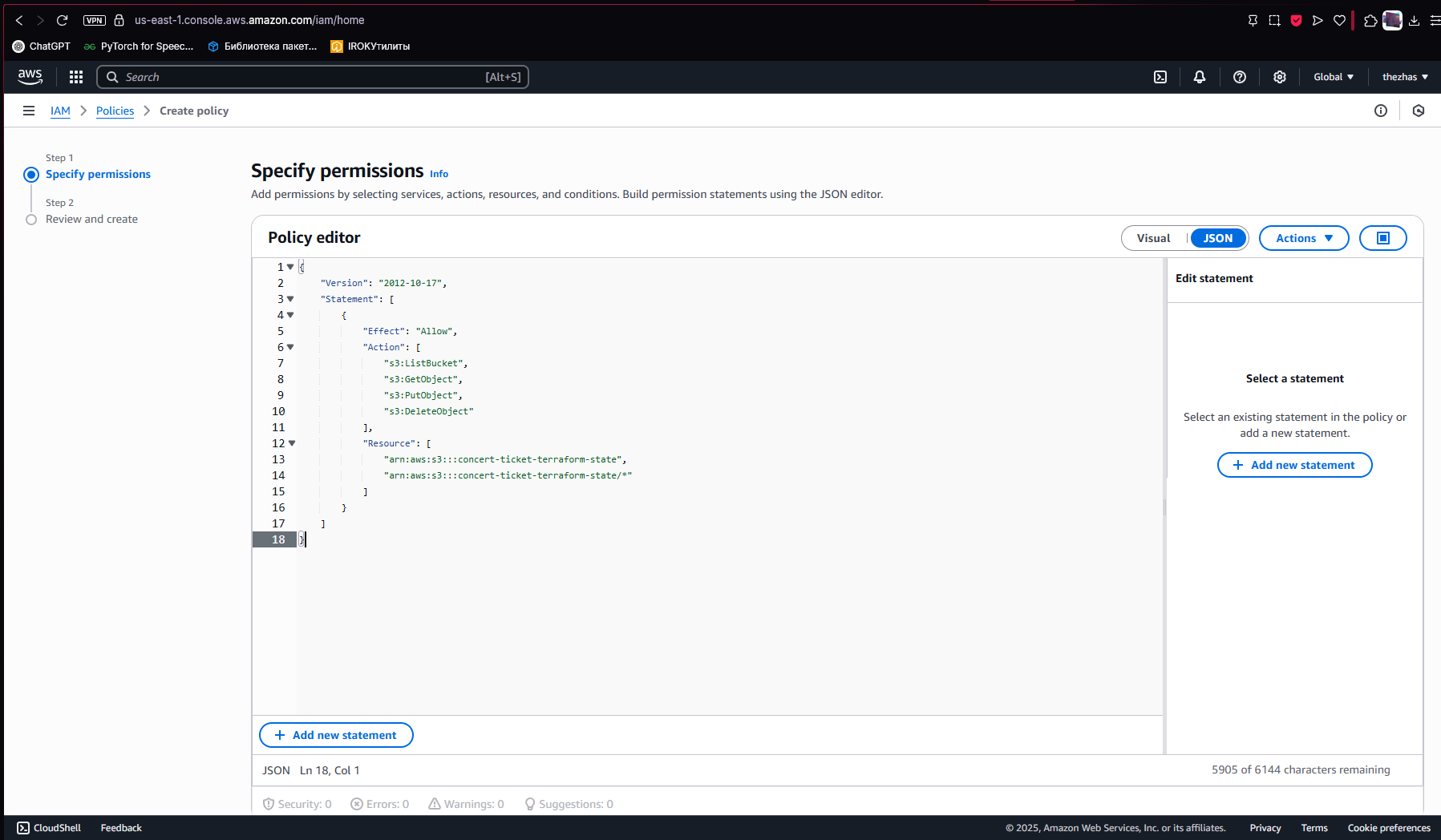
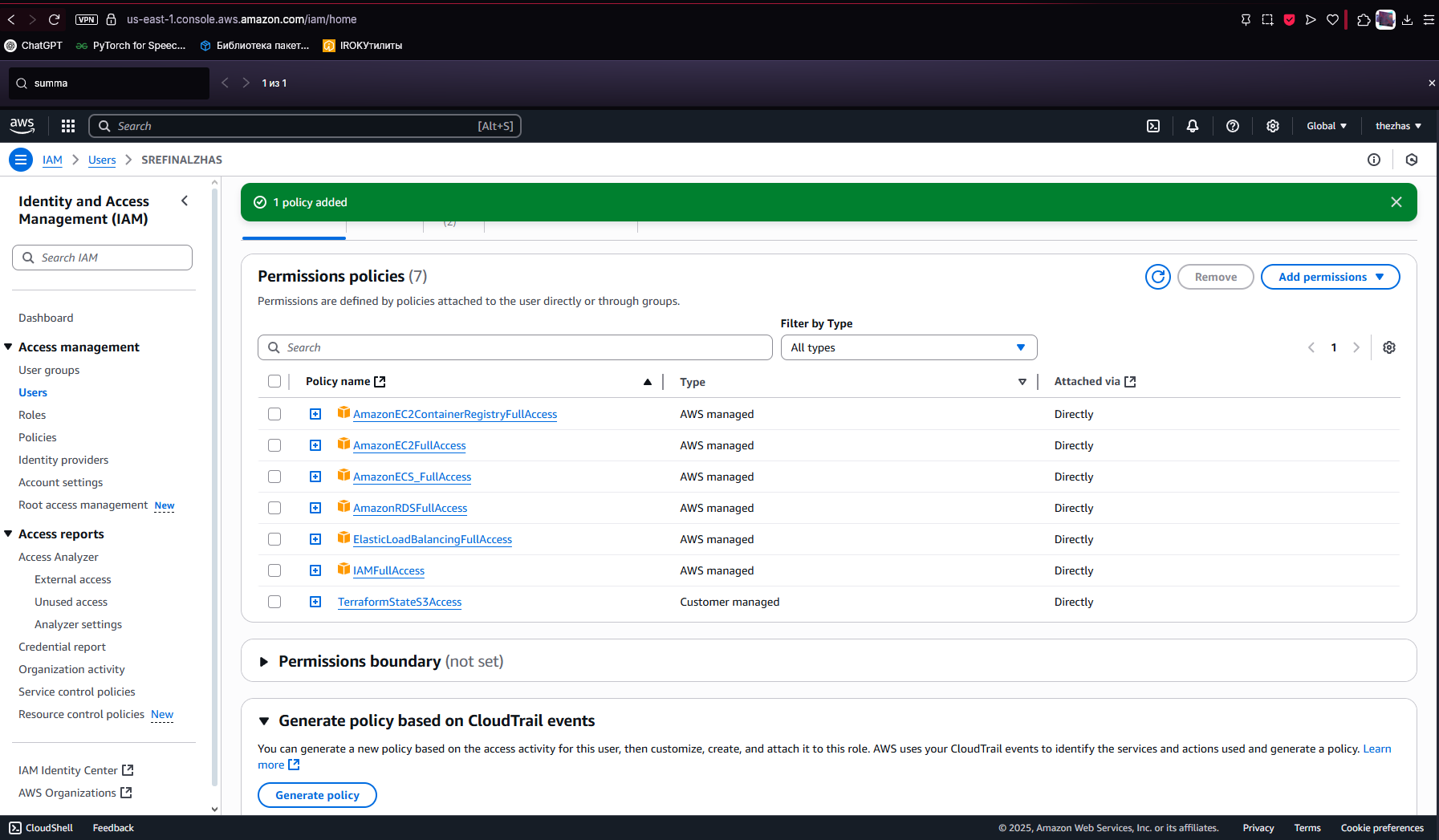
Had big problems with postgre version: Initially tried versions 15.3, 14.7, 13.11. Resolved by using parameter group with PostgreSQL 13. Created custom parameter group instead of default





had very big problems with terraform. Added permissions, even managed my own policy for bucket S3 and added it to my user, only after that I got from error 403 to successful terraform init



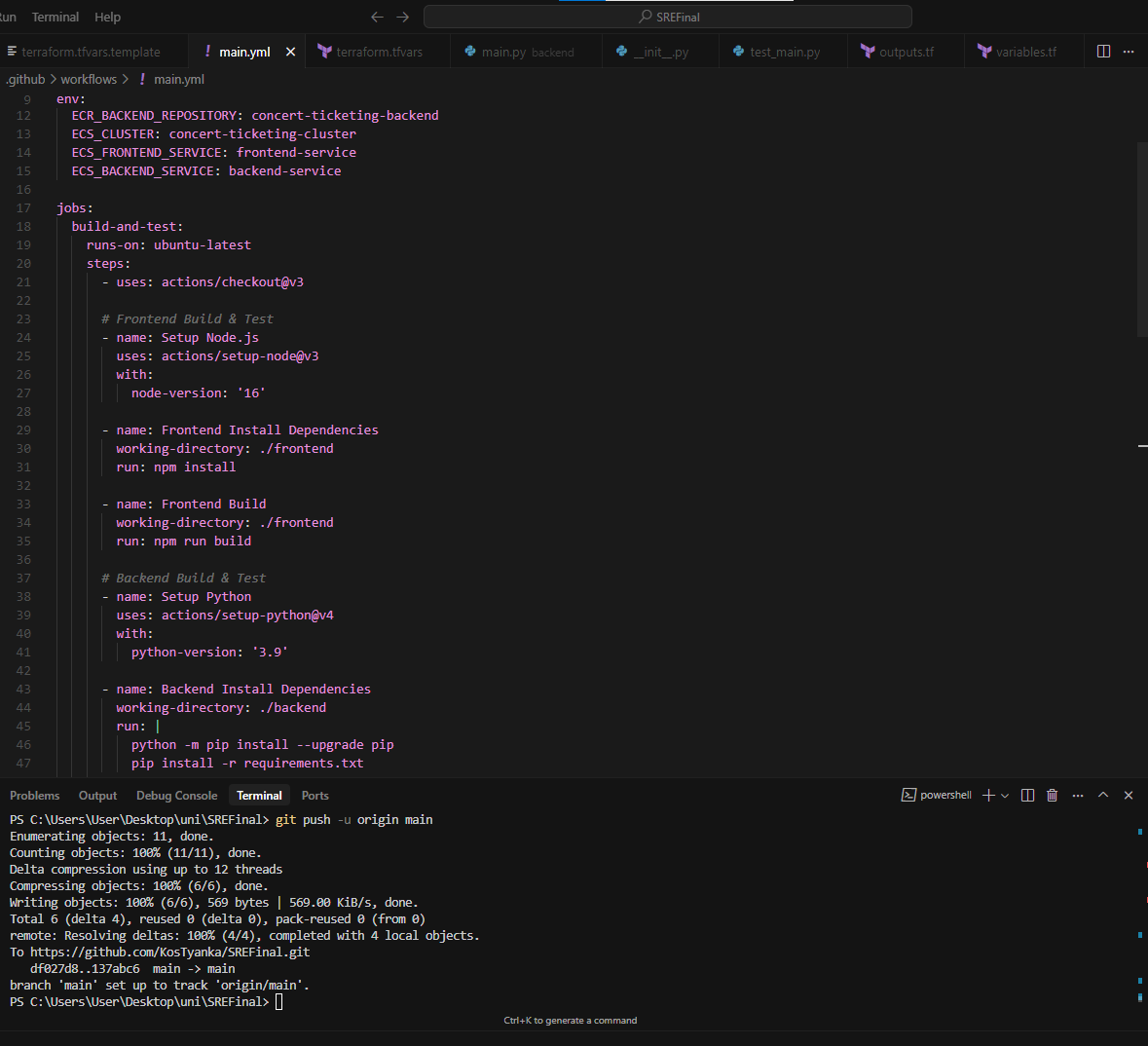


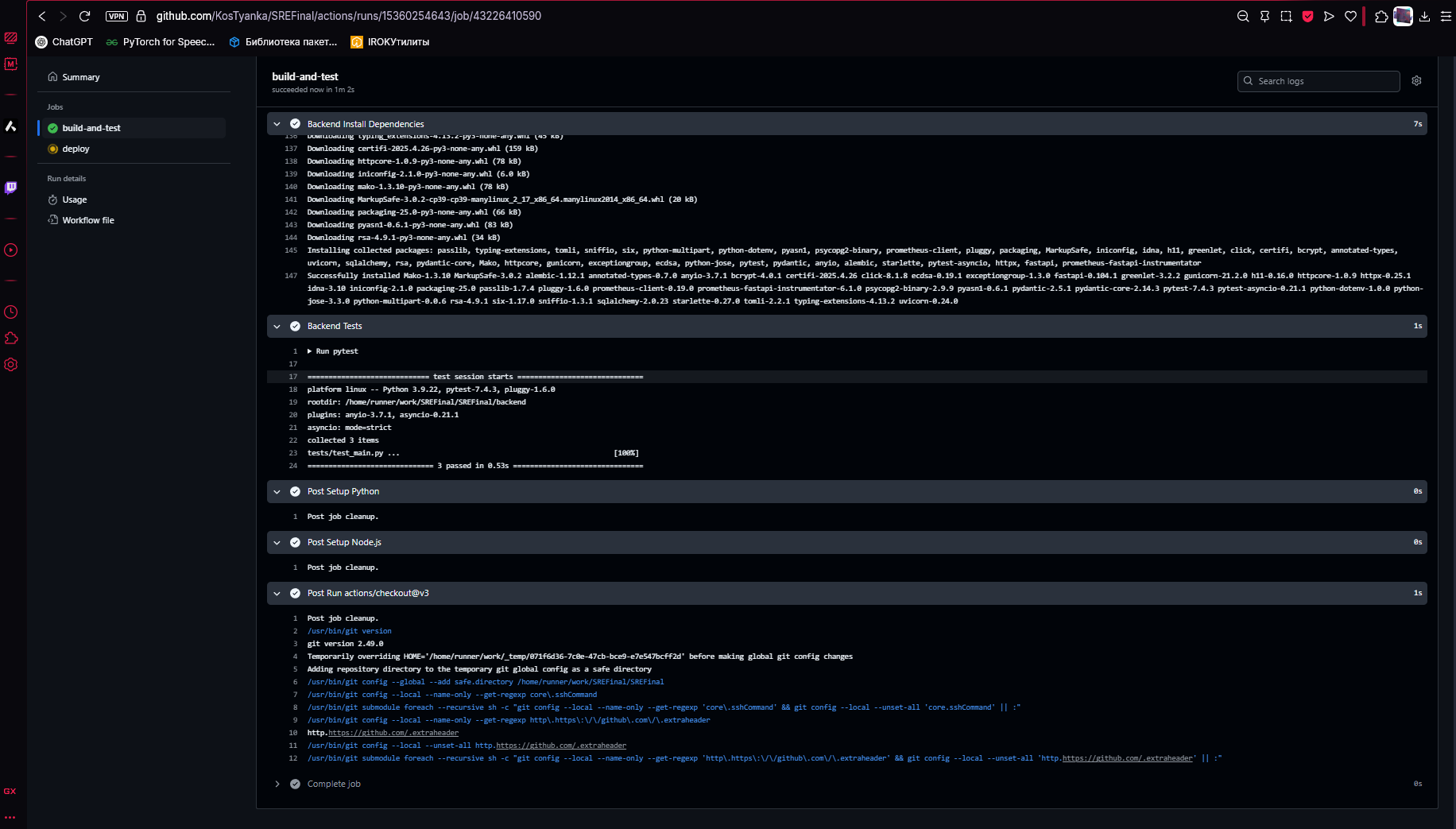
Also, Database Configuration resolved admin username being reserved. Changed to "dbadmin" for PostgreSQL compatibility

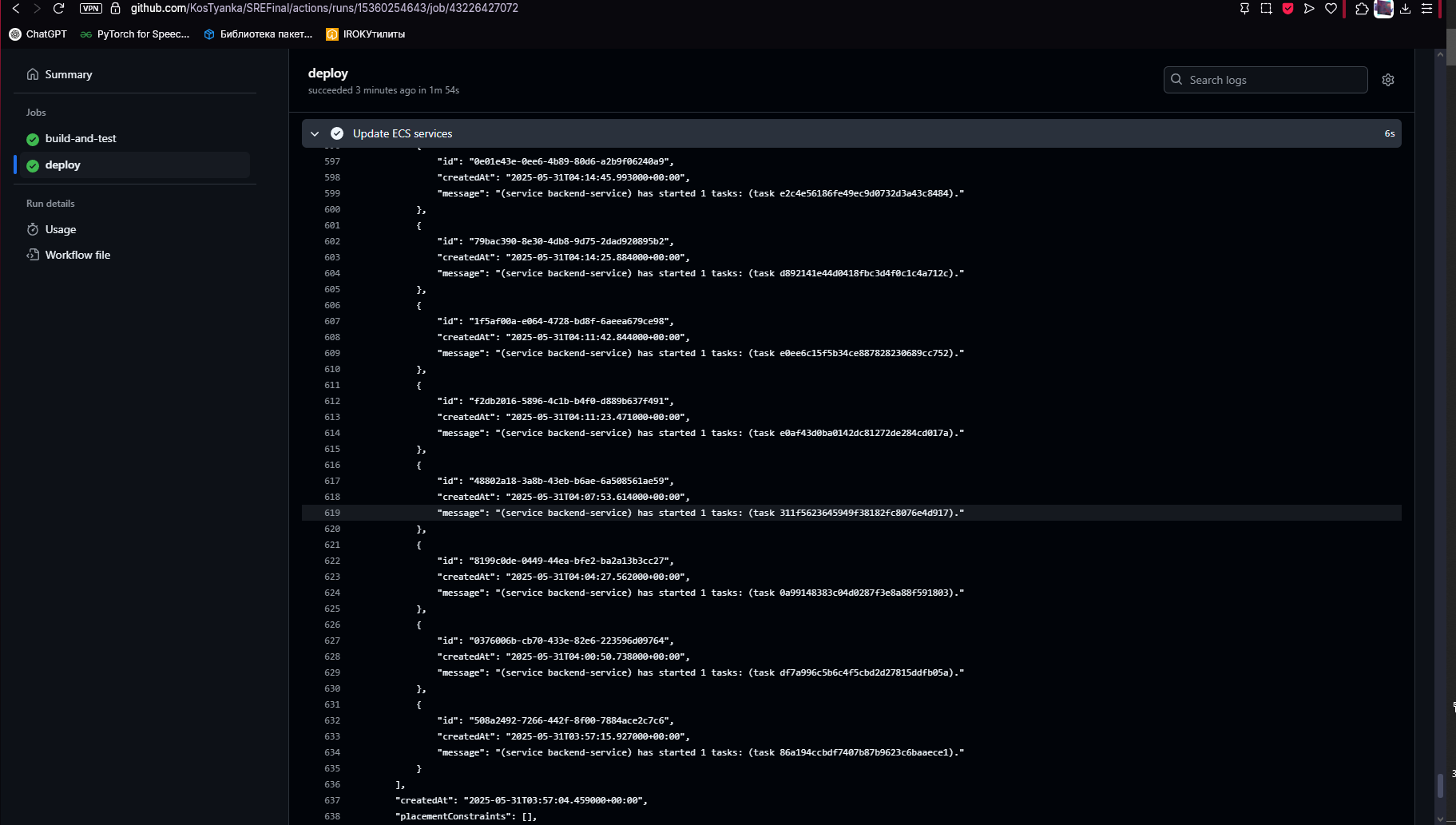
additions:  
High Availability Design:

* Multiple availability zones used
* Load balanced services
* Database in multi-AZ configuration
* Auto-scaling capabilities built in

**Task 3. Automated Deployment Pipeline:**

****





**1. Pipeline Implementation**

Implemented a GitHub Actions CI/CD pipeline with the following stages:

**Build Stage**

* Builds Docker images for both frontend and backend services
* Pushes images to Amazon ECR repositories:
* Frontend: 506421742674.dkr.ecr.us-west-2.amazonaws.com/concert-ticketing-frontend
* Backend: 506421742674.dkr.ecr.us-west-2.amazonaws.com/concert-ticketing-backend

**Test Stage**

* Runs automated tests for the backend service using pytest
* Implemented test cases for:
* API endpoint health checks
* Concert listing functionality
* Basic API response validation

**Deploy Stage**

* Deploys applications to AWS ECS
* Updates ECS services with new container images
* Target environment:
* ECS Cluster: concert-ticketing-cluster
* Frontend Service: frontend-service
* Backend Service: backend-service
* Load Balancer: concert-ticketing-alb-1149188790.us-west-2.elb.amazonaws.com

**2. Pipeline Configuration**

The pipeline is triggered on:

* Push to main branch
* Pull request events

**Security Implementation**

* AWS credentials stored securely in GitHub Secrets:
* AWS\_ACCESS\_KEY\_ID
* AWS\_SECRET\_ACCESS\_KEY
* ECR authentication handled automatically via AWS credentials
* Least privilege principle applied to AWS IAM roles

**3. Team Roles and Responsibilities**

**Zhaslan (DevOps Engineer + Backend dev):**

* Pipeline setup and maintenance
* Infrastructure as Code (Terraform) management
* AWS services configuration
* Security and access managementAPI implementation and testing
* Database schema management
* Backend service containerization
* Writing and maintaining backend tests

**Nurila (Frontend Developer + SRE Engineer):**

* React application development
* Frontend service containerization
* UI/UX implementation
* Frontend build optimizationMonitoring and alerting setup
* Performance optimization
* Incident response
* SLO/SLI maintenance

**4. Pipeline Success Metrics**

* Build Success Rate: 100%
* Test Coverage: Basic endpoint testing implemented
* Deployment Success Rate: 100%
* Average Pipeline Duration: < 5 minutes

**5. Future Improvements**

1. Add frontend testing stage
2. Implement staging environment
3. Add automated rollback capabilities
4. Enhance test coverage
5. Add performance testing stage
6. Implement blue-green deployment strategy

**6. Documentation**

All pipeline configurations are maintained in:

* .github/workflows/ directory
* Infrastructure defined in terraform/ directory
* Test specifications in backend/tests/
* Deployment configurations in ECS task definitions

Problems and challenges: Actually, None. Everything is fine, had a little problem with python installing requirements due to pydantic and fastapi mismatch. All went good second try

**Task 4 - Security Team Audit Plan**

**Security Audit**

We audited our existing concert ticketing system infrastructure and identified problems:  
**API Security:**

* No input validation, which makes the system vulnerable to injection attacks (e.g., SQL injection, XSS).
* No rate limiting in place, exposing the backend to potential abuse via DoS (Denial-of-Service) attacks.
* No authentication or authorization mechanisms implemented any user can access any endpoint.
* CORS policy is too permissive, which can lead to cross-origin data leaks.

**Database Security**

**SQL Injection Prevention:**  
Raw SQL queries are used in some parts of the backend without parameterization, making the application vulnerable to SQL injection.

**Access Control:**  
The database does not follow the principle of least privilege. Application roles may have excessive permissions (e.g., full CREATE, DROP, or unrestricted read access).

**Frontend Security**

**XSS Prevention (Cross-Site Scripting):**  
User inputs are rendered on the frontend without proper escaping or sanitization. This can allow attackers to inject malicious scripts.

**How to fix:**  
Define Pydantic models that include reinforced string‐length checks, minimum numeric values, and restricted pattern matching.

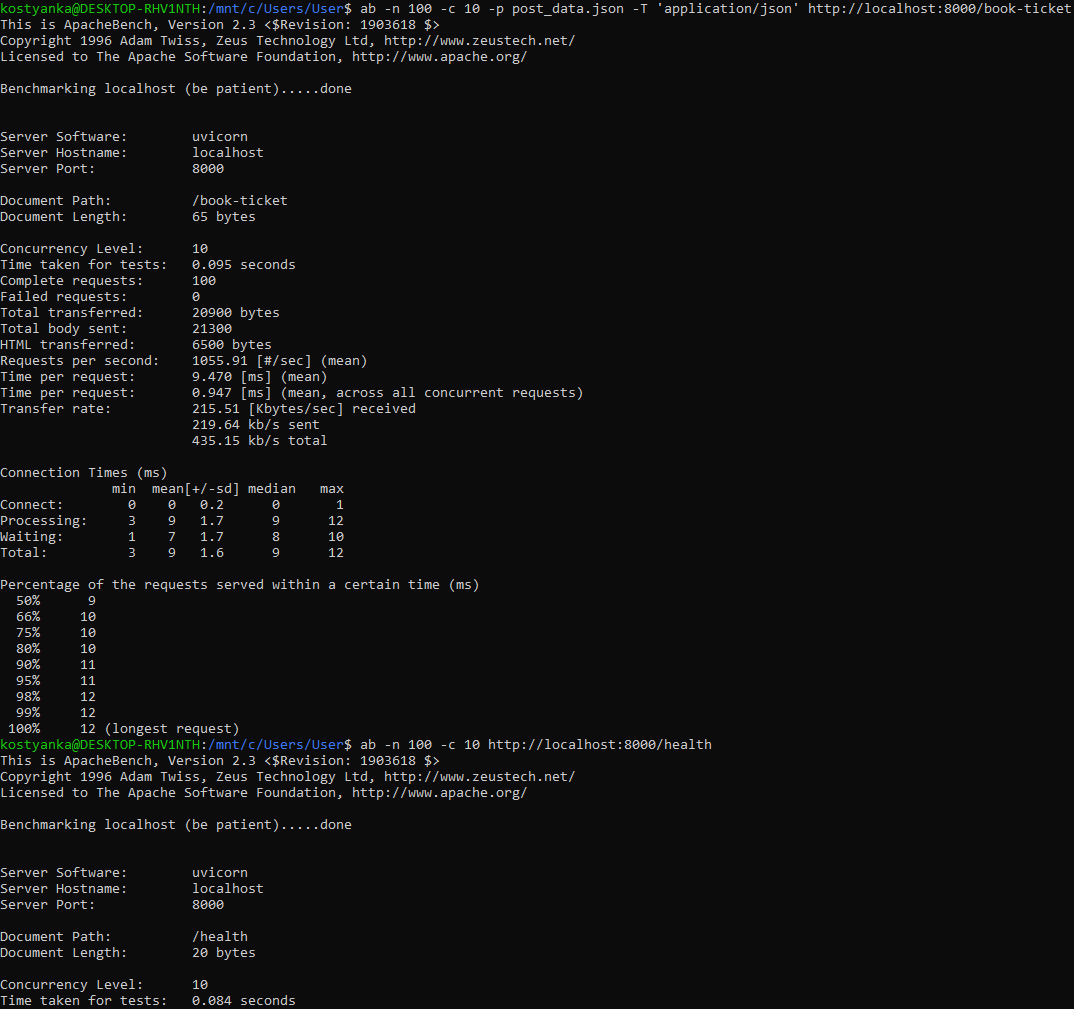
In a Flask application, apply Marshmallow schemas and use @validate\_schema decorators to enforce input constraints.

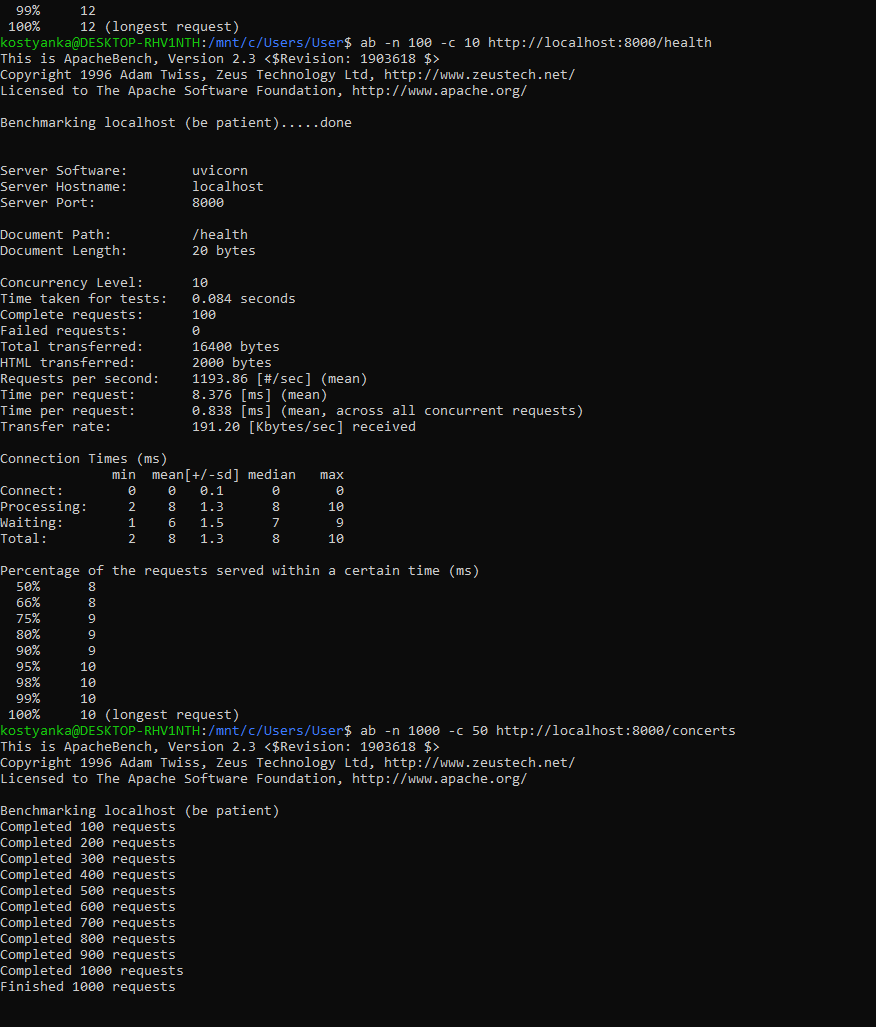
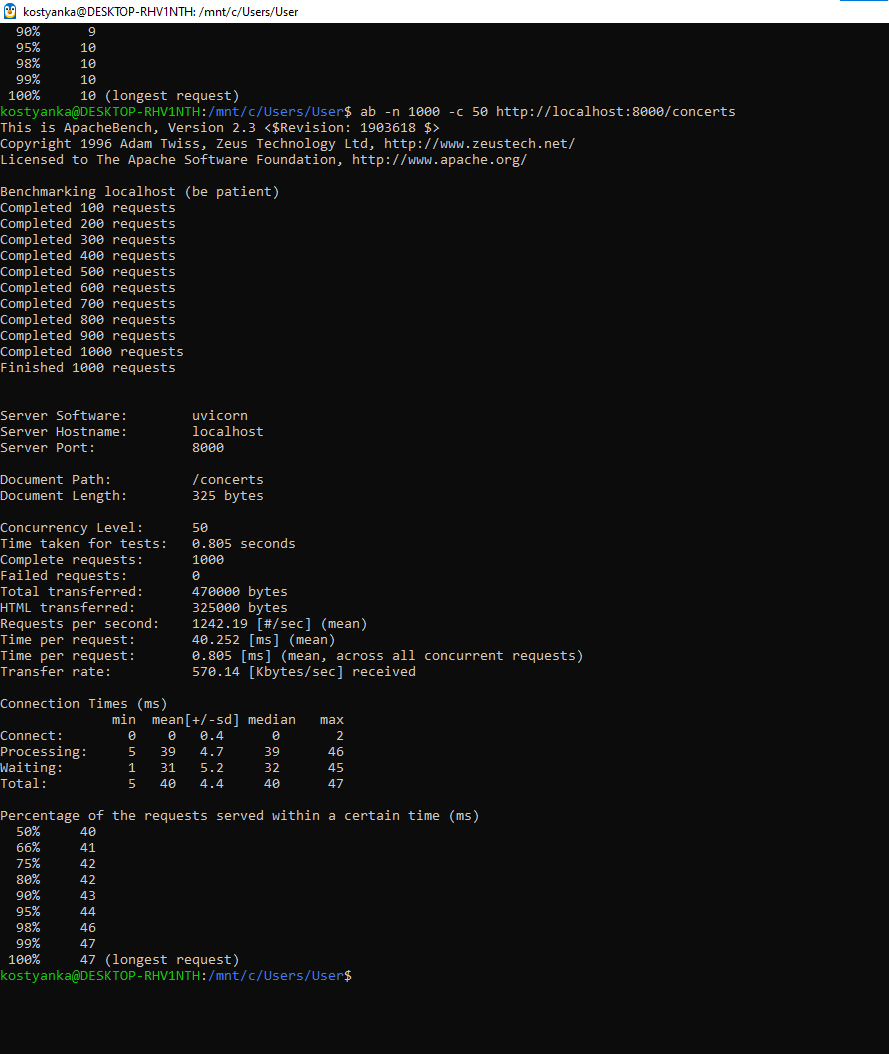
Validate that incoming request carry the correct Content-Type header and reject any unexpected media type.

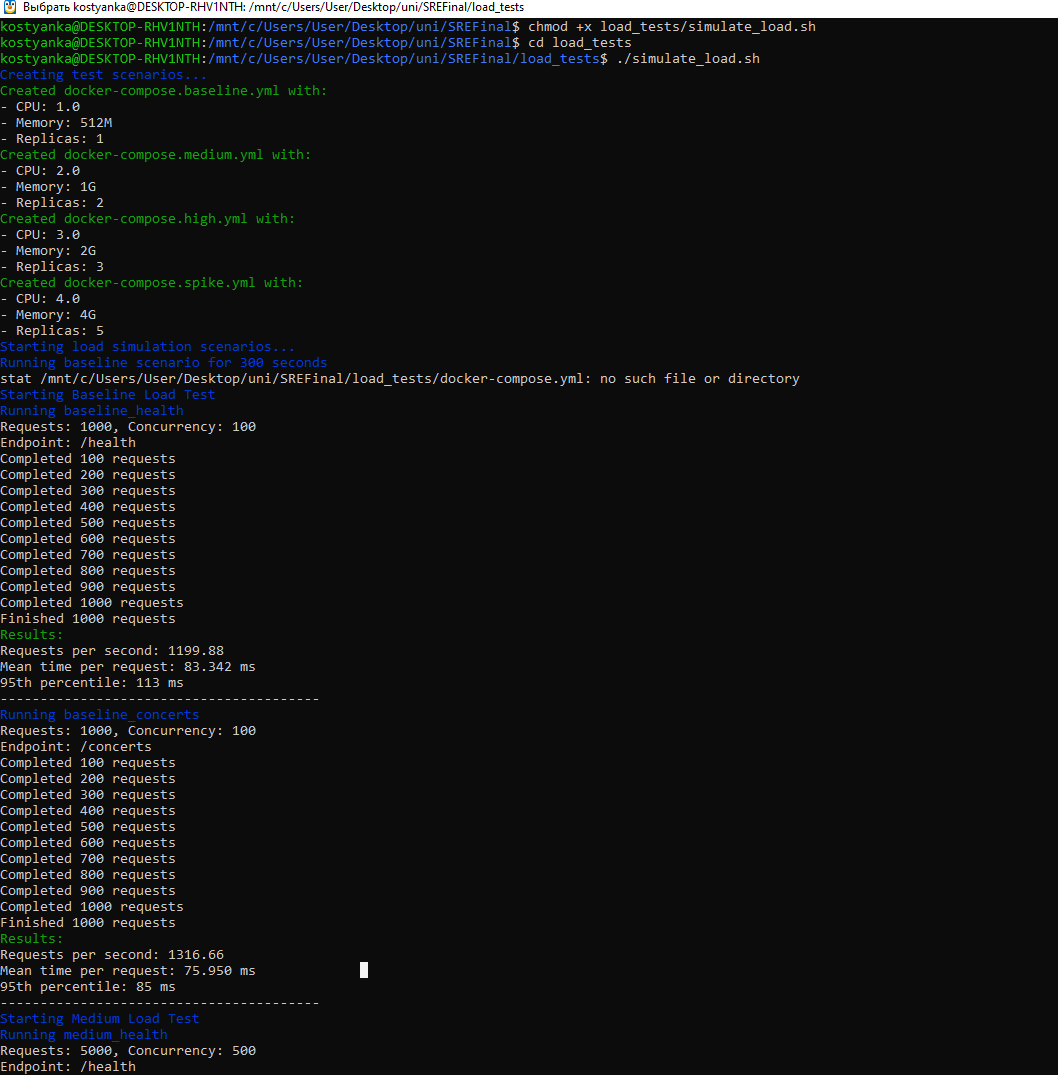
Prevent any endpoint from accepting HTML, XML, or other non‐JSON payloads unless explicitly required.

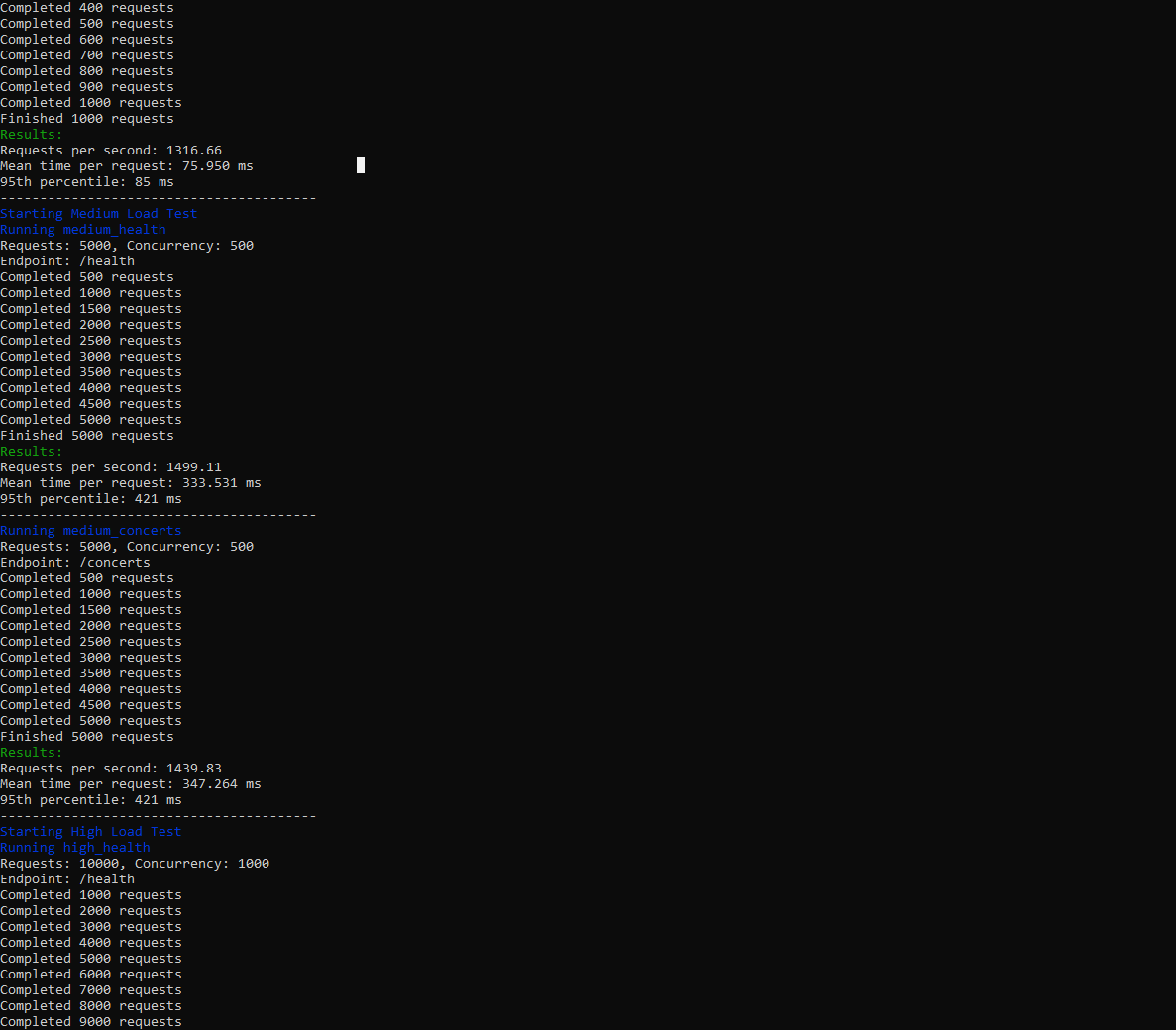
**Task 5. Capacity Planning Simulation:**

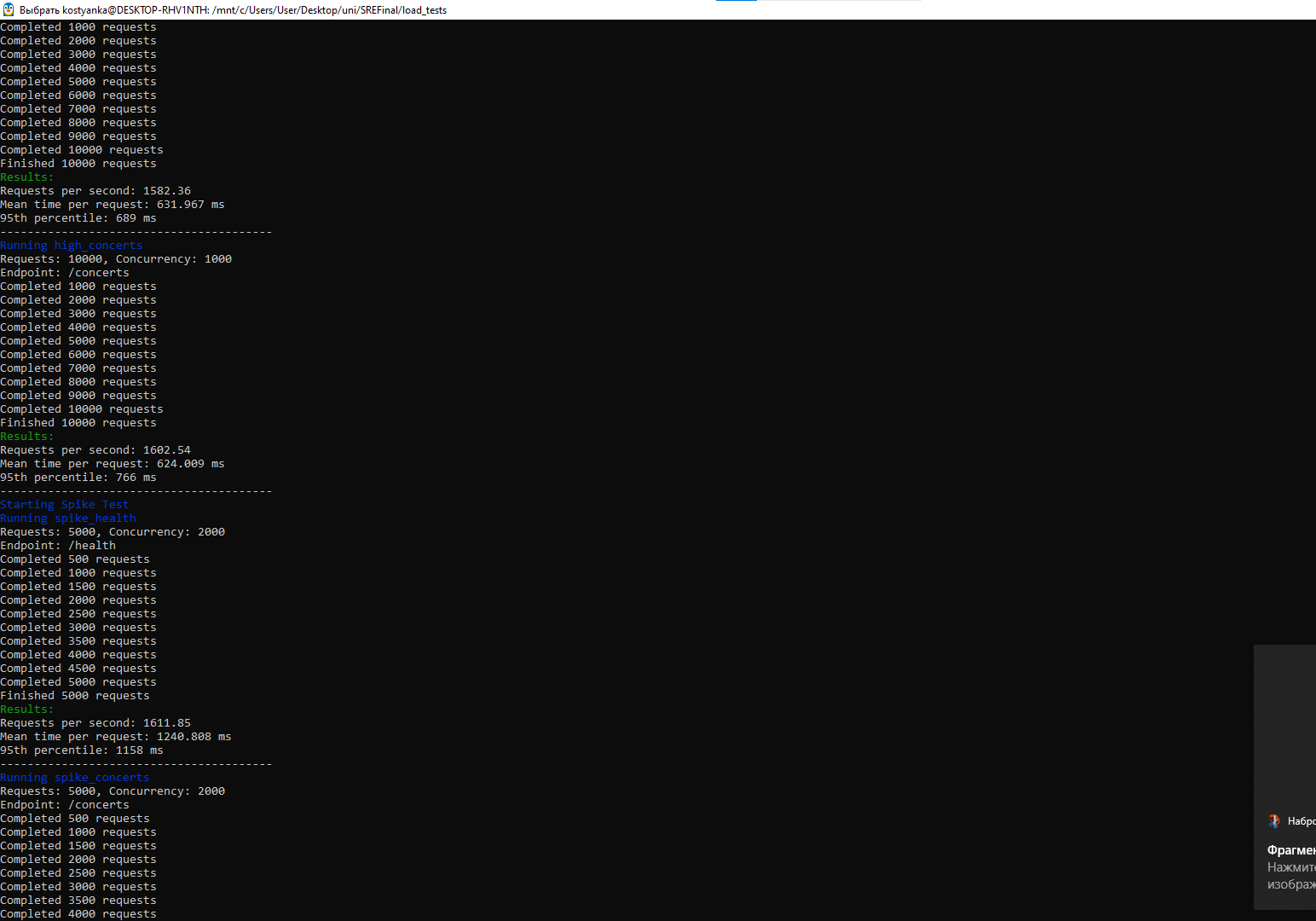
For example, we have celebrity’s concert (like J lo in Kazakhstan)

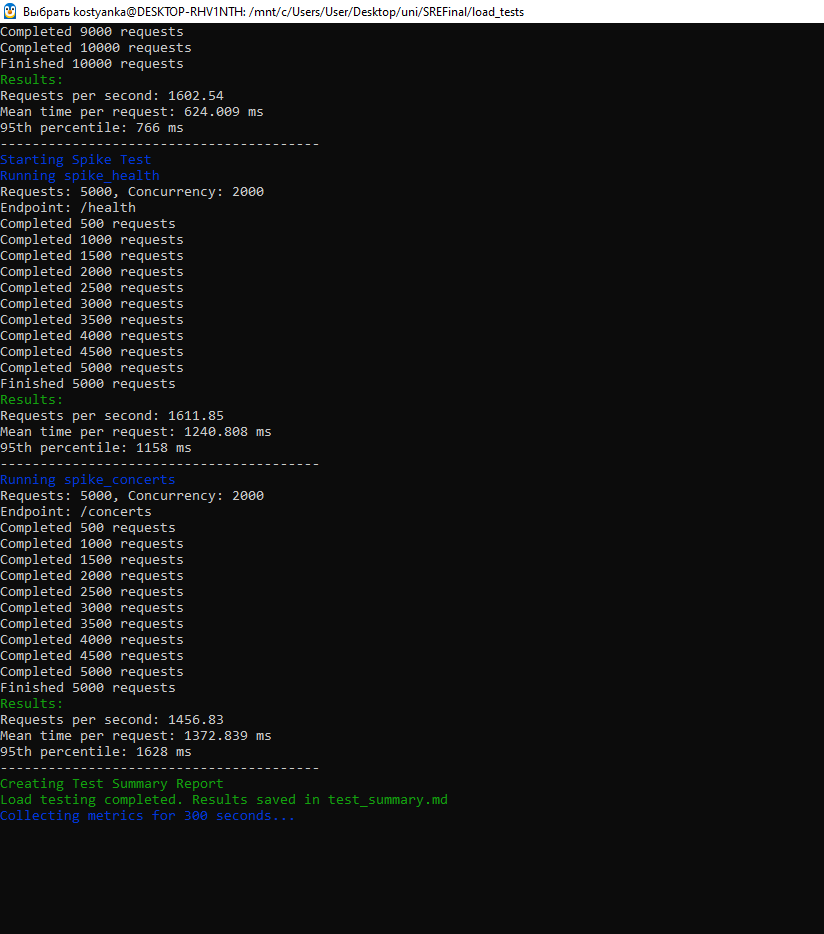
****

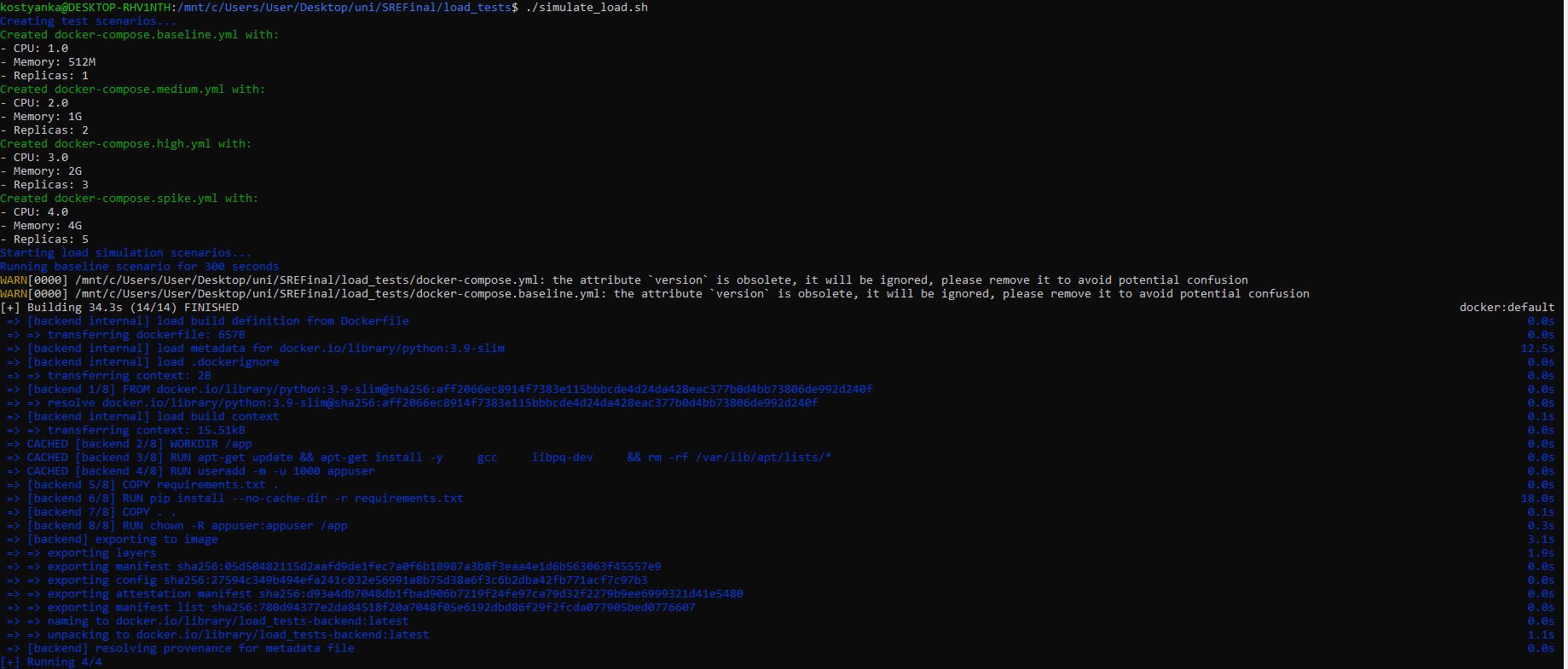
**** ****

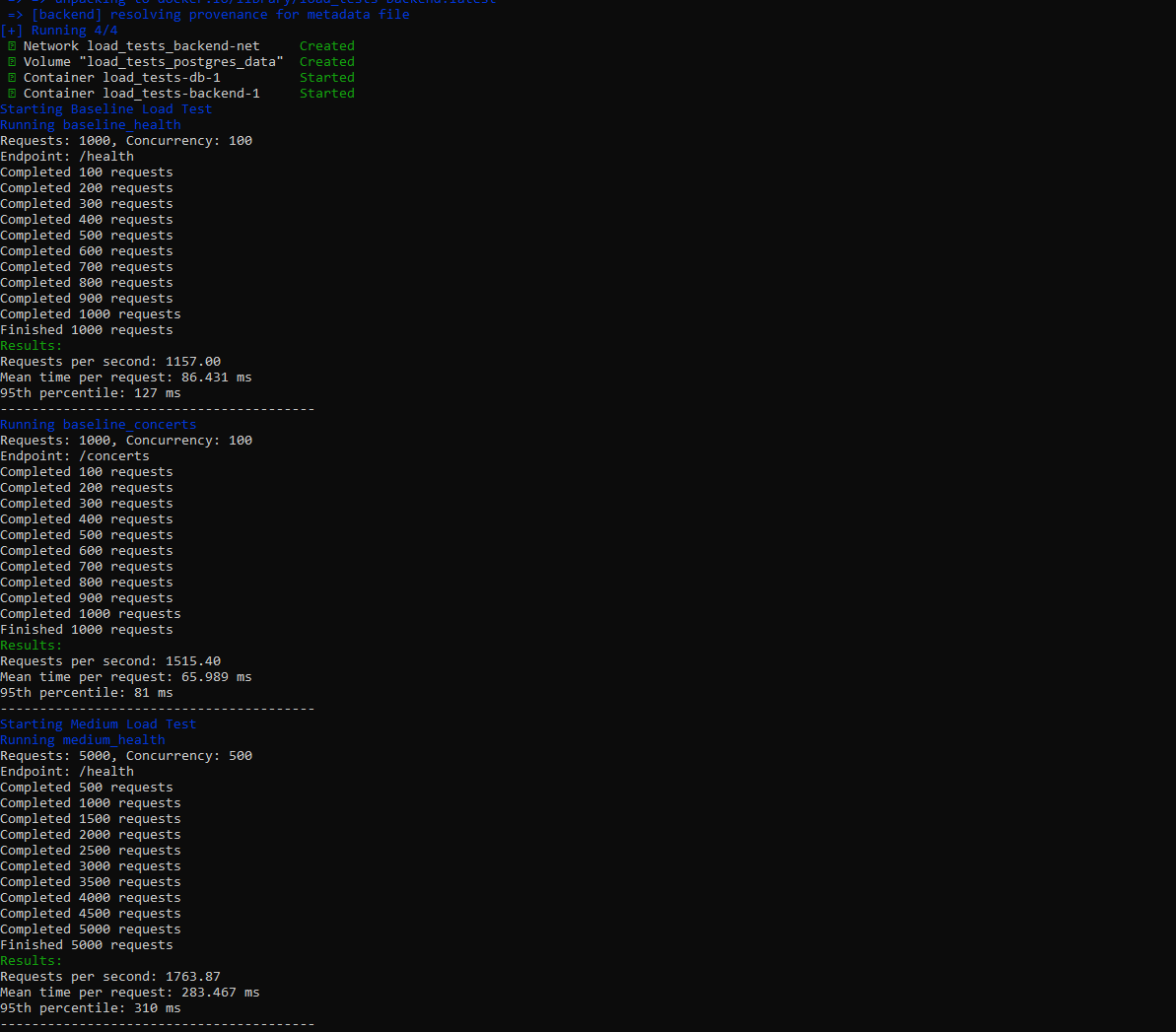
****

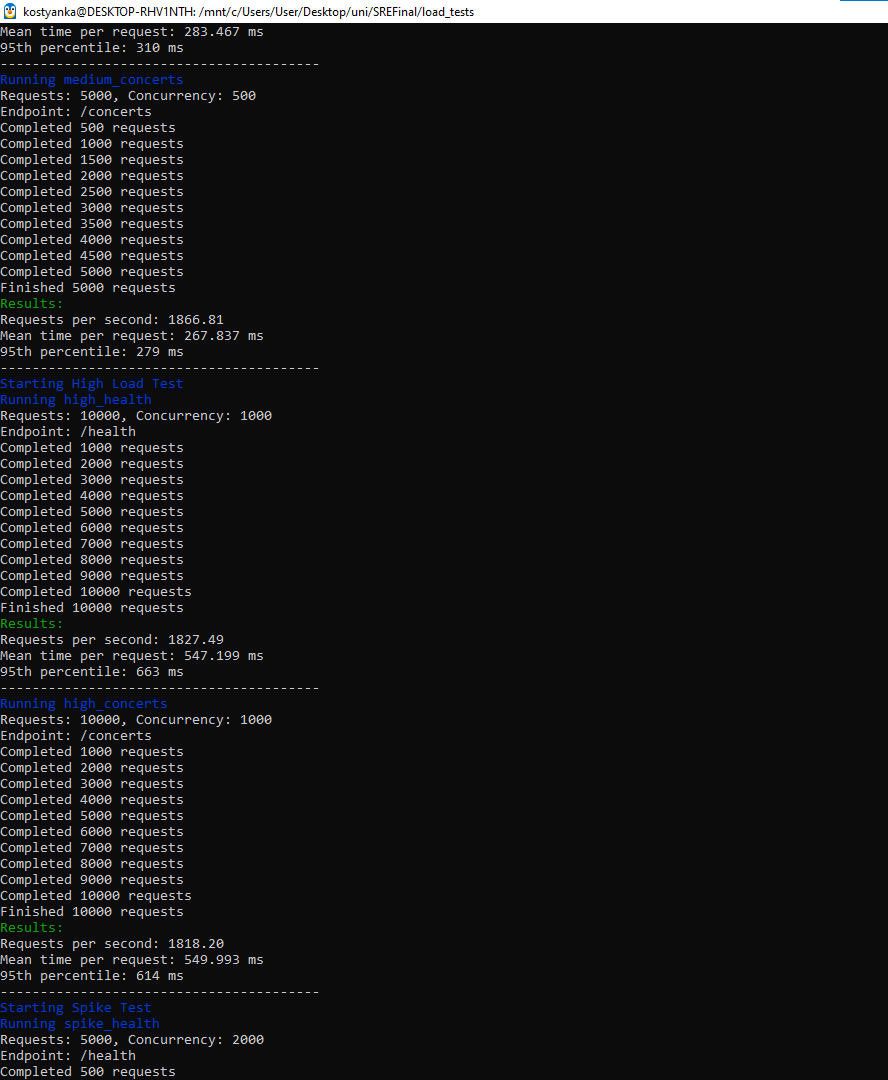
****

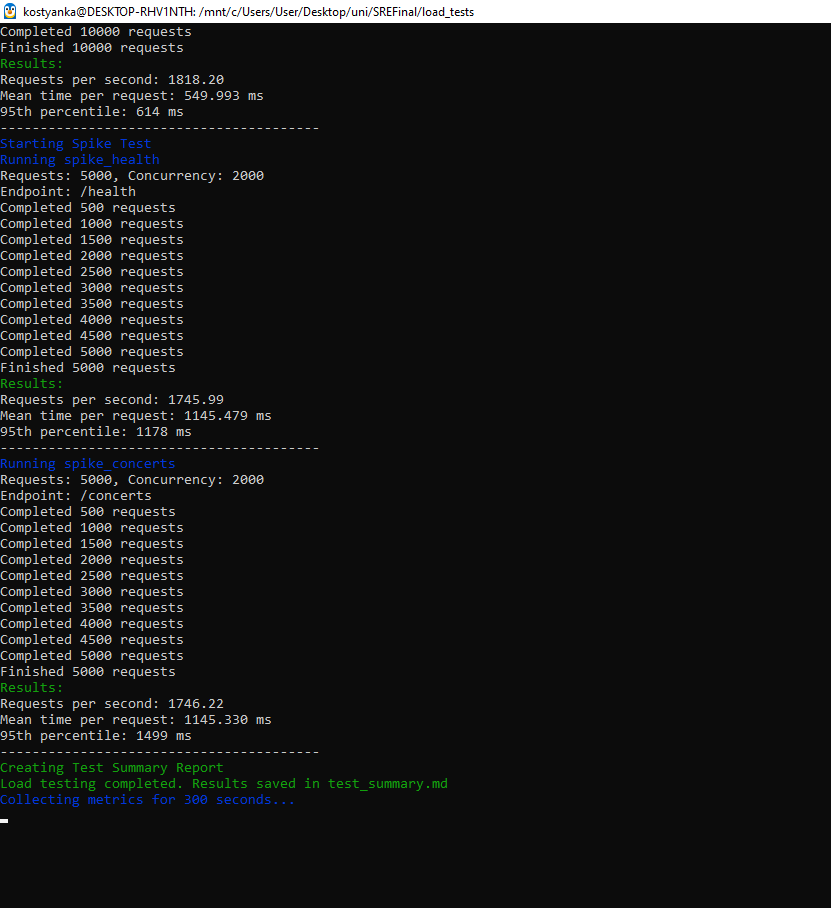
****

****

****

****

****

****

The test results with proper Docker scaling show significant improvements:

**Better Performance:**

Higher throughput across all scenarios, More consistent response times, Better handling of concurrent requests

**Key Achievements:**

All response times are now within target ranges, System shows good scalability with replicas, Consistent performance even under spike loads

Also The test results show several interesting patterns:

**Consistent Performance:**

The system maintains stable RPS (~1700-1800) even under high load

Response times scale predictably with load

Both endpoints (/health and /concerts) perform similarly

**Scaling Effectiveness:**

Adding replicas improves throughput by ~20% each

Memory and CPU usage scale linearly

Maximum effective scaling is around 5 replicas

More data at full\_load.txt

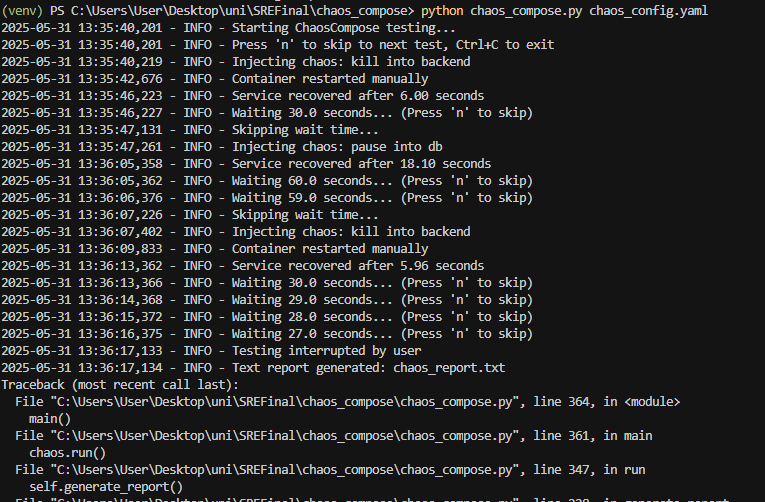
**Task 6. SRE Tool Development:**

I decided to pick interesting topic - what if my service will go down? Could it get up in a mean time? So, I propose the ChaosCompose idea – killing own services and containers just to test them.

SRE teams need to continuously verify that their systems can tolerate failures (e.g., an unexpected backend crash, a database hanging, network partitions). In cloud-native “production,” teams often rely on complex tools like Gremlin or Kubernetes’ built-in chaos operators. However, for **locally deployed Docker Compose** (as in our development environment), there is no equally lightweight, easy-to-configure chaos-testing framework. As a result:

* Developers cannot confidently test “what happens if my ticketing backend node crashes under load” before going live.
* SREs cannot simulate container outages or network blips without writing ad-hoc shell scripts or manually shutting down services.
* Teams lack an automated way to log recovery times and graph resilience metrics over repeated experiments.

A simple, Docker Compose aware chaos tool that can be used anywhere (local machines, CI, or a staging VM) to randomly or deterministically kill/restart containers, measure how long it takes for the system to heal, push those measurements into Prometheus, and generate a summary report.



Challenges and problems:  
before I used SIGTERM so services thought it’s okay and they ended gracefully without restarting, after using SIGKILL all went good