I. Overall System Architecture

The system will consist of the following main components:

- AWS EKS (Elastic Kubernetes Service): Manages the K8S cluster to run containers.
- AWS ALB (Application Load Balancer): Balances load for Pods.
- AWS Auto Scaling Group (ASG): Automatically scales worker nodes based on resource demands.
- Amazon RDS or DynamoDB: Database for the application.
- Amazon S3: Storage for static files or data.
- AWS CloudWatch & Prometheus: Monitoring and logging.
- AWS IAM: Access management.
- AWS VPC & Subnet: Private network for security and connectivity.

II. Detailed Deployment Steps

Step 1: Create a Kubernetes Cluster on AWS (EKS)

1. Install AWS CLI and kubectl on the local machine:

```
aws configure
```

2. Create an EKS cluster using AWS CLI:

```
eksctl create cluster --name my-cluster --region us-east-1 --nodegroup-name worker-nodes --node-type t3.medium --nodes 3
```

3. Configure kubectl to connect to the EKS cluster:

```
aws eks update-kubeconfig --region us-east-1 --name my-cluster
```

Step 2: Package the Application into a Container and Push it to Amazon ECR (Elastic Container Registry)

1. Write a Dockerfile for the application.
2. Build and push the image to Amazon ECR:

```
aws ecr create-repository --repository-name my-app
docker build -t my-app .
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin <aws-account-id>.dkr.ecr.us-east-1.amazonaws.com
```

docker tag my-app:latest <aws-account-id>.dkr.ecr.us-east-1.amazonaws.com/my-app:latest
docker push <aws-account-id>.dkr.ecr.us-east-1.amazonaws.com/my-app:latest

Step 3: Deploy the Application to Kubernetes

1. Create Deployment and Service:
   - Write a `deployment.yaml` file.

2. Deploy to Kubernetes:

```
kubectl apply -f deployment.yaml
```

Step 4: Configure Horizontal Pod Autoscaler (HPA)

1. Create HPA to auto-scale pods based on CPU usage:

```
kubectl autoscale deployment my-app --cpu-percent=50 --min=2 --max=10
```

2. Check the autoscaler status:

```
kubectl get hpa
```

Step 5: Configure Auto Scaling Group (ASG) for Worker Nodes

1. Adjust ASG settings in the EKS cluster:

```
eksctl scale nodegroup --cluster=my-cluster --name=worker-nodes --nodes-min=2 --nodes-max=10
```

2. Enable Cluster Autoscaler:
   - Install Helm Chart for Cluster Autoscaler:

```
helm repo add autoscaler https://kubernetes.github.io/autoscaler
helm repo update
helm install cluster-autoscaler autoscaler/cluster-autoscaler --set autoDiscovery.clusterName=my-cluster
```

Step 6: System Optimization for Performance Requirements

- Throughput: 500 requests per second
- Response Time: p99 response time of <100ms

6.1 Choose the Right Worker Node Size

- Use AWS EC2 C6i or M6i instances (CPU-optimized, memory-balanced).
- Scale node groups as needed:

```
eksctl scale nodegroup --cluster=my-cluster --name=worker-nodes --nodes-min=2 --nodes-max=10
```

6.2 Load Balancer & Traffic Control

- Use AWS ALB (Application Load Balancer):
  - Distributes traffic evenly.
  - Supports HTTP/2 and WebSockets to reduce overhead.
- Kubernetes Ingress Controller: Use AWS ALB Ingress Controller for optimized routing.
- Connection Pooling: Reduce open connections using `keep-alive`.

6.3 Configure Horizontal Pod Autoscaler (HPA)

- Autoscale based on CPU & Memory to maintain response time <100ms.

```
kubectl autoscale deployment my-app --cpu-percent=50 --min=2 --max=10
```

- Cluster Autoscaler: If pod count increases but no available nodes exist, Cluster Autoscaler adds nodes automatically.

6.4 Application Optimization

6.4.1 Reduce Response Time (<100ms)

- Caching:
  - Use Redis or Memcached for static data or frequent queries.
  - Utilize AWS ElastiCache to reduce database load.
- Connection Pooling:
  - Use connection pooling (PgBouncer for PostgreSQL) to limit new database connections.
- Reduce API Calls:
  - Use GraphQL or gRPC instead of REST to reduce request count.
  - Batch multiple requests instead of sending many small ones.
- Optimize Database Queries:
  - Use Indexing on critical tables.
  - Avoid N+1 queries using JOIN or ORM optimizations.

6.4.2 Prevent Overload

- Rate Limiting:
  - Limit requests per IP using AWS API Gateway or Istio.
- Circuit Breaker:
  - Stop requests temporarily if services fail to prevent overload.

Step 7: Monitoring & Logging

7.1 Application Performance Monitoring (APM)

- Use Prometheus + Grafana to monitor CPU, Memory, and Latency.
- Use AWS CloudWatch Logs to track system errors.

7.2 Alerting & Auto-Healing

- Use AWS Lambda + CloudWatch to restart containers if response time >100ms continuously.

Step 8: Overall System Architecture Diagram