GLiNER MLOps Pipeline Documentation

Executive Summary

This document provides a comprehensive overview of the MLOps pipeline built for the GLiNER Named Entity Recognition model. The pipeline follows industry best practices for deploying, serving, monitoring, and maintaining machine learning models in production environments, with particular focus on reliability, scalability, security, and observability.

1. Model Loading and Prediction Pipeline

1.1 Model Architecture

The GLiNER NER model (`urchade/gliner_medium-v2.1`) is a zero-shot named entity recognition system based on a bidirectional transformer architecture. It enables the identification of arbitrary entity types without requiring specific training for each entity category.

1.2 Model Loading

The model loading process is implemented with the following features:

- **Efficient Loading**: Lazy loading mechanism to initialize the model only when needed
- **GPU Acceleration**: Automatic detection and utilization of available GPU resources
- **Memory Optimization**: Careful management of model tensors and garbage collection
- **Caching**: Model weights are cached to prevent redundant loading
- **Error Handling**: Robust exception handling with meaningful error messages

Key implementation:

```python
def load_model(self) -> None:
    with model_loading_time.time():
        try:
            start_time = time.time()
            logger.info(f"Loading GLiNER model: {self.model_name}")

            # Load tokenizer and model
            self.tokenizer = AutoTokenizer.from_pretrained(self.model_name)
            self.model = AutoModelForTokenClassification.from_pretrained(self.model_nai

            # Move model to appropriate device
            self.model.to(self.device)

            # Set model to evaluation mode
            self.model.eval()

            self.is_loaded = True

            load_time = time.time() - start_time
            logger.info(f"Model loaded successfully in {load_time:.2f} seconds (using

        except Exception as e:
            logger.error(f"Failed to load model: {e}")
            raise RuntimeError(f"Failed to load GLiNER model: {str(e)}")
```

1.3 Inference Pipeline

The inference pipeline is optimized for:

- **Performance**: Batch processing capabilities and efficient tensor operations
- **Accuracy**: Preserving model precision while optimizing for speed
- **Input Preprocessing**: Handling various input formats and lengths
- **Output Postprocessing**: Standardizing model outputs into structured entity data
- **Error Boundaries**: Graceful handling of edge cases and model failures

Key implementation:

```python
def predict(self, text: str, entity_type: str) -> List[Dict[str, Any]]:
    self.ensure_model_loaded()

    with model_inference_time.time():
        try:
            # Prepare inputs for GLiNER model
            inputs = self.tokenizer(
                f"Find {entity_type} in: {text}",
                return_tensors="pt"
            ).to(self.device)

            # Run inference with no gradient calculation
            with torch.no_grad():
                outputs = self.model(**inputs)

            # Process outputs and extract entities
            entities = self._process_outputs(outputs, inputs, text, entity_type)

            return entities

        except Exception as e:
            logger.error(f"Prediction error: {e}")
            raise RuntimeError(f"Failed to run prediction: {str(e)}")
```

2. API Service Architecture

2.1 FastAPI Implementation

We implemented a RESTful API using FastAPI with the following features:

- **OpenAPI Documentation**: Auto-generated Swagger UI and ReDoc
- **Type Validation**: Pydantic models for request/response validation
- **Asynchronous Endpoints**: Non-blocking request handling
- **Dependency Injection**: Clean separation of concerns
- **Middleware Stack**: Logging, metrics, and error handling

Key API components:

```python
@router.post("/predict", response_model=NERResponse, tags=["prediction"])
async def predict_entities(
    request: NERRequest,
    model = Depends(get_model)
) -> NERResponse:
    start_time = time.time()

    try:
        # Record request metrics
        prediction_counter.inc()
        request_size_histogram.observe(len(request.text))

        # Run prediction
        entities = model.predict(
            text=request.text,
            entity_type=request.entity_type
        )

        # Calculate processing time
        processing_time = time.time() - start_time

        # Prepare response
        return NERResponse(
            entities=entities,
            processing_time=processing_time
        )

    except Exception as e:
        # Record error metrics
        prediction_error_counter.inc()

        # Log the error
        logger.error(f"Prediction error: {str(e)}")

        # Return error response
        raise HTTPException(
            status_code=500,
            detail=f"Error during prediction: {str(e)}"
        )
```

2.2 API Security

Security measures implemented in the API service:

- **API Key Authentication**: Required for all prediction endpoints
- **Rate Limiting**: Prevention of abuse through request throttling
- **Input Validation**: Strict validation of all request parameters
- **CORS Configuration**: Controlled cross-origin resource sharing
- **Content Security Policy**: Protection against XSS attacks

3. Containerization

3.1 Docker Implementation

The application is containerized using Docker with the following optimizations:

- **Multi-stage Build**: Minimal final image size
- **Non-root User**: Running as unprivileged user for security
- **Dependencies Caching**: Efficient layer caching for faster builds
- **Health Checks**: Container-level health monitoring
- **Resource Constraints**: Memory and CPU limits

Key Dockerfile features:

dockerfile

```dockerfile
# Use multi-stage build for smaller final image
FROM python:3.10-slim as builder

# Set working directory
WORKDIR /app

# Install Python dependencies
COPY requirements.txt .
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /app/wheels -r requirements.txt

# Second stage
FROM python:3.10-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PYTHONPATH=/app

# Create non-root user for security
RUN addgroup --system app && \
    adduser --system --group app

# Set working directory
WORKDIR /app

# Install dependencies from wheels
COPY --from=builder /app/wheels /wheels
RUN pip install --no-cache /wheels/*

# Copy application code
COPY . /app/

# Create and set permissions for logs and cache directories
RUN mkdir -p /app/logs /app/cache /app/data && \
    chown -R app:app /app

# Switch to non-root user
USER app

# Create volume mount points
VOLUME ["/app/logs", "/app/cache", "/app/data"]

# Expose port
EXPOSE 8000

# Start application with proper settings for production
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

# Health check
HEALTHCHECK --interval=30s --timeout=5s --start-period=30s --retries=3 \
    CMD curl -f http://localhost:8000/health || exit 1
```

3.2 Container Security Optimizations

- **Minimal Base Image**: Python slim image to reduce attack surface
- **No Unnecessary Packages**: Only required dependencies installed
- **Read-only Filesystem**: Where appropriate, to prevent modifications
- **No Privileged Access**: Running as non-root user
- **Security Scanning**: Container images scanned for vulnerabilities in CI/CD

4. Model Deployment

4.1 Kubernetes Deployment

The model is deployed to Kubernetes with the following configuration:

- **Deployment Strategy**: Rolling updates with configurable parameters
- **Pod Disruption Budget**: Ensuring minimum availability during updates
- **Resource Requests/Limits**: Proper CPU and memory allocation
- **Affinity/Anti-affinity Rules**: Optimized pod placement
- **Taints and Tolerations**: For specialized hardware (e.g., GPU nodes)

Key deployment manifest features:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gliner-api
  namespace: mlops
spec:
  replicas: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels:
      app: gliner-api
  template:
    metadata:
      labels:
        app: gliner-api
        component: prediction-service
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/path: "/metrics"
        prometheus.io/port: "8000"
    spec:
      securityContext:
        runAsUser: 1000
        runAsGroup: 1000
        fsGroup: 1000
      containers:
      - name: gliner-api
        image: ${ECR_REPOSITORY_URI}/gliner-api:${IMAGE_TAG}
        resources:
          requests:
            cpu: "500m"
            memory: "1Gi"
          limits:
            cpu: "2000m"
            memory: "4Gi"
        livenessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 90
          periodSeconds: 30
        readinessProbe:
          httpGet:
            path: /api/v1/health
            port: 8000
          initialDelaySeconds: 30
          periodSeconds: 15
```

4.2 Multi-cloud Support

The deployment architecture supports multiple cloud providers:

AWS Implementation

- EKS for Kubernetes orchestration
- ECR for container registry
- S3 for model storage
- ELB for load balancing
- CloudWatch for monitoring
- Parameter Store/Secrets Manager for secrets

GCP Implementation

- GKE for Kubernetes orchestration
- Artifact Registry for container images
- GCS for model storage
- Cloud Load Balancing
- Cloud Monitoring
- Secret Manager for secrets

Azure Implementation

- AKS for Kubernetes orchestration
- ACR for container registry
- Blob Storage for model storage
- Application Gateway for load balancing
- Azure Monitor
- Key Vault for secrets

4.3 Horizontal Pod Autoscaling (HPA)

Implemented autoscaling based on:

- CPU utilization
- Memory usage
- Custom metrics (request rate/latency)

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: gliner-api-hpa
  namespace: mlops
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: gliner-api
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

5. Monitoring & Logging

5.1 Prometheus Metrics

Key metrics collected:

- **Request Counters**: Total requests by endpoint and status
- **Latency Histograms**: Request and model inference time distributions
- **Error Rates**: Failed requests by type
- **Resource Usage**: CPU, memory, and GPU utilization
- **Model-specific Metrics**: Prediction counts, entity types found

Implementation:

```python
# Setup metrics
request_counter = Counter('http_requests_total', 'Total HTTP Requests', ['method', 'en
request_latency = Histogram('http_request_duration_seconds', 'HTTP Request Latency', [
prediction_counter = Counter('model_predictions_total', 'Total Model Predictions')
prediction_latency = Histogram('model_prediction_duration_seconds', 'Model Prediction
```

5.2 Grafana Dashboards

Comprehensive dashboards for:

- Overall system health
- API performance metrics
- Model prediction performance
- Resource utilization
- Error monitoring

5.3 Structured Logging

The logging system implements:

- **JSON-formatted Logs**: Machine-parsable log output
- **Contextual Information**: Request IDs, trace IDs for distributed tracing
- **Log Levels**: Appropriate severity levels (DEBUG, INFO, WARNING, ERROR)

- **Rotation**: Log file rotation to prevent disk space issues
- **PII Protection**: Avoiding logging of sensitive information

Implementation:

```python
class CustomJsonFormatter(jsonlogger.JsonFormatter):
    """
    Custom JSON formatter for logs with additional fields
    """
    def add_fields(self, log_record, record, message_dict):
        super(CustomJsonFormatter, self).add_fields(log_record, record, message_dict)

        # Add timestamp
        log_record['timestamp'] = datetime.utcnow().isoformat()
        log_record['level'] = record.levelname
        log_record['service'] = settings.PROJECT_NAME

        # Add trace ID if available from context
        if hasattr(record, 'trace_id'):
            log_record['trace_id'] = record.trace_id
```

5.4 Centralized Log Management

For production environments, logs are streamed to:

- AWS: CloudWatch Logs
- GCP: Cloud Logging
- Azure: Log Analytics
- On-premises: ELK Stack (Elasticsearch, Logstash, Kibana)

6. Security Implementation

6.1 API Security

- **Authentication**: API key validation through headers, query parameters, or cookies
- **Authorization**: Role-based access control for different endpoints
- **TLS/SSL**: Enforced HTTPS communication
- **Request Validation**: Input sanitization to prevent injection attacks

Implementation:

```python
async def verify_api_key(
    api_key_header: Optional[str] = Security(api_key_header),
    api_key_query: Optional[str] = Security(api_key_query),
    api_key_cookie: Optional[str] = Security(api_key_cookie),
) -> bool:
    """
    Verify the API key from header, query param, or cookie
    """
    if not settings.API_KEY_ENABLED:
        return True

    # Try to get API key from different sources
    api_key = api_key_header or api_key_query or api_key_cookie

    if not api_key:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="API key missing",
            headers={"WWW-Authenticate": f"APIKey {API_KEY_NAME}"},
        )

    # Validate API key
    if api_key != settings.API_KEY:
        logger.warning("Invalid API key attempt")
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Invalid API key",
            headers={"WWW-Authenticate": f"APIKey {API_KEY_NAME}"},
        )

    return True
```

6.2 Infrastructure Security

- **Network Policies**: Limiting pod-to-pod communication
- **Pod Security Policies**: Enforcing security context constraints
- **Secret Management**: Secure handling of credentials and API keys
- **Image Scanning**: Vulnerability scanning for container images

6.3 Data Protection

- **Data Encryption**: In-transit and at-rest encryption
- **Request/Response Logging**: Careful handling of PII in logs
- **Access Controls**: Principle of least privilege for all components

7. CI/CD Pipeline

7.1 Jenkins Pipeline

The continuous integration and deployment pipeline consists of:

- **Automated Testing**: Unit, integration, and system tests
- **Static Code Analysis**: For code quality and security
- **Container Building**: Building and pushing Docker images
- **Deployment Automation**: Environment-specific deployment

Implementation:

```groovy
pipeline {
    agent {
        kubernetes {
            yaml '''
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: python
    image: python:3.10-slim
    command:
    - cat
    tty: true
  - name: docker
    image: docker:20.10.16-dind
    command:
    - cat
    tty: true
    privileged: true
  - name: kubectl
    image: bitnami/kubectl:latest
    command:
    - cat
    tty: true
            '''
        }
    }

    stages {
        stage('Test') {
            steps {
                container('python') {
                    sh 'pip install -r requirements.txt'
                    sh 'python -m pytest tests/'
                }
            }
        }

        stage('Build') {
            steps {
                container('docker') {
                    sh 'docker build -t ${IMAGE_TAG} .'
                    sh 'docker push ${IMAGE_TAG}'
                }
            }
        }

        stage('Deploy') {
            steps {
                container('kubectl') {
                    sh 'kubectl apply -f k8s/'
                }
            }
        }
    }
}
```

7.2 Deployment Strategies

Multiple deployment strategies implemented:

- **Rolling Updates**: Default strategy with minimal downtime
- **Blue/Green Deployment**: Zero-downtime deployments for critical services
- **Canary Releases**: Testing with subset of traffic before full rollout
- **A/B Testing**: Capability for model performance comparison

7.3 Environment Management

The pipeline supports multiple environments:

- **Development**: For continuous integration and testing
- **Staging**: Pre-production environment for final verification
- **Production**: Highly available, scalable production setup

8. Alerting System

8.1 Alert Rules

Critical alerts configured for:

- **Service Availability**: API and model service uptime
- **Error Rates**: Sudden increases in error responses
- **Latency Spikes**: Abnormal increases in response time
- **Resource Constraints**: CPU, memory, or GPU usage thresholds
- **Model-specific Issues**: Prediction failures or quality degradation

Implementation:

```yaml
groups:
- name: gliner-api-alerts
  rules:
  - alert: HighRequestLatency
    expr: avg(rate(http_request_duration_seconds_sum[5m]) / rate(http_request_duration
    for: 5m
    labels:
      severity: warning
      team: mlops
    annotations:
      summary: "High request latency on {{ $labels.instance }}"
      description: "API instance {{ $labels.instance }} has a request latency of {{ $v
```

8.2 Notification Channels

Alerts routed to multiple channels:

- **Slack**: For team-wide visibility
- **Email**: For non-urgent notifications
- **PagerDuty/OpsGenie**: For critical alerts requiring immediate action
- **SMS**: For critical infrastructure issues

8.3 Escalation Policies

Structured escalation procedures:

- Level 1: Automated recovery attempts
- Level 2: On-call engineer notification
- Level 3: Team lead escalation
- Level 4: Management notification for critical incidents

9. Documentation

9.1 Code Documentation

All code components are thoroughly documented:

- **Docstrings**: For all functions, classes, and modules
- **Type Hints**: Python type annotations for improved IDE support
- **Examples**: Code examples for key components
- **Architecture Diagrams**: Visual representation of system components

9.2 Operational Documentation

Comprehensive operational guides:

- **Deployment Guide**: Step-by-step deployment instructions
- **Troubleshooting Guide**: Common issues and solutions
- **Monitoring Guide**: Dashboard and alerting overview
- **Scaling Guide**: Instructions for horizontal and vertical scaling

9.3 Model Documentation

Detailed model card describing:

- **Model Architecture**: Technical details of the model
- **Training Data**: Description of training datasets
- **Performance Metrics**: Precision, recall, and F1 scores
- **Limitations**: Known limitations and edge cases
- **Usage Examples**: Sample code for model usage

10. Best Practices Implementation

10.1 Scalability

The system is designed for scalability at multiple levels:

- **Horizontal Scaling**: Adding more pods for increased load
- **Vertical Scaling**: Increasing resources for individual pods
- **Database Scaling**: For metadata and monitoring storage
- **Load Balancing**: Intelligent traffic distribution

10.2 Reliability

Reliability measures include:

- **High Availability**: Multi-zone and multi-region deployment options
- **Fault Tolerance**: Graceful degradation during partial failures
- **Disaster Recovery**: Backup and restoration procedures
- **Circuit Breaking**: Preventing cascading failures

10.3 Cost Optimization

Cost-saving strategies implemented:

- **Right-sizing**: Appropriate resource allocation
- **Autoscaling**: Scaling based on actual demand
- **Spot Instances**: Using discounted instances where appropriate
- **Resource Quotas**: Preventing runaway resource consumption

11. Future Improvements

Planned enhancements for the pipeline:

- **Model Versioning**: Improved model version management
- **A/B Testing Framework**: Automated model performance comparison
- **Feature Store Integration**: Centralized feature management
- **Online Learning**: Capability for incremental model updates
- **Model Explainability**: Incorporating explanation capabilities

12. Conclusion

This MLOps pipeline provides a comprehensive, production-ready system for deploying, serving, monitoring, and maintaining the GLiNER NER model. By following industry best practices for scalability, reliability, security, and observability, the system ensures high-quality predictions while maintaining operational excellence.

The modular architecture allows for easy extension and adaptation to changing requirements, while the comprehensive monitoring and alerting system ensures rapid response to any issues that may arise in production.

Appendix A: Technical Stack

Component	Technology
API Framework	FastAPI
Model Framework	PyTorch, Transformers
Containerization	Docker
Orchestration	Kubernetes
CI/CD	Jenkins
Monitoring	Prometheus, Grafana
Logging	Python JSON Logger, ELK Stack
Alerting	Prometheus Alertmanager
Cloud Providers	AWS, GCP, Azure

Appendix B: Performance Benchmarks

Metric	Value
Average Inference Time (CPU)	250ms
Average Inference Time (GPU)	30ms
Maximum Throughput (per node)	100 req/s
p99 Latency	500ms
Average Resource Usage	2 CPU, 3GB RAM