Enterprise AI Infrastructure - System Integration Guide

Complete integration patterns, end-to-end examples, and production deployment strategies for the unified enterprise AI platform.

Table of Contents

Overview

The Enterprise AI Infrastructure consists of four interconnected modules that work together to provide a complete, production-ready AI platform:

- **Enterprise Logger**: High-performance logging with async batching (100K+ logs/sec)
- **Analytics Platform**: Real-time data processing, visualization, and reporting
- **Knowledge Graph Engine**: Advanced graph algorithms and semantic reasoning
- **Attention Engine**: State-of-the-art transformer implementations

Why Integration Matters

While each module is powerful individually, their true strength emerges when working together:

```
Data Flow: Raw Data → Attention Processing → Knowledge Graph → Analytics →
Reports
Logging: All operations logged with correlation tracking across modules
```

Architecture & Data Flow

High-Level Architecture

```
┌──────────────────────────────┐     ┌──────────────────────────────┐
┌──────────────────────────────┐
│   Data Sources    │     │    API Gateway    │     │  Client Apps     │
│                   │     │                   │     │                  │
│ • Databases       │◄───►│ • REST APIs       │◄───►│ • Web UI          │
│ • APIs            │     │ • GraphQL         │     │ • Mobile Apps    │
│ • Files           │     │ • WebSockets      │     │ • Dashboards     │
└──────────────────────────────┘     └──────────────────────────────┘
└──────────────────────────────┘
         │                        │                        │
         ▼                        ▼                        ▼
┌──────────────────────────────────────────────────────────────────────────────────────
│                    ENTERPRISE AI PLATFORM                           │
├──────────────────────────────────────────────────────────────────────────────────────
│   ┌──────────────────────────┐   ┌──────────────────────────┐
┌──────────────────────────┐              │
│   │   ATTENTION     │   │   KNOWLEDGE     │   │   ANALYTICS     │        │
│   │    ENGINE       │   │     GRAPH       │   │   PLATFORM      │        │
│   │                 │   │    ENGINE       │   │                 │        │
│   │ • Flash Attn    │   │ • Graph Algs    │   │ • Processing    │        │
│   │ • RoPE/ALiBi    │   │ • Reasoning     │   │ • Visualization│        │
│   │ • Multi-Frmk    │   │ • ML Integ.     │   │ • Reporting     │        │
│   └──────────────────────────┘   └──────────────────────────┘
└──────────────────────────┘              │
│            │                    │                    │             │
│            └───────────────────────────────┴───────────────────────────────┘
│
│                                 │                                 │
│
┌──────────────────────────────────────────────────────────────────────────────────────
│
│   │                ENTERPRISE LOGGER                         │    │
│   │                                                          │    │
│   │ • Async Batching    • Performance Monitoring            │    │
│   │ • Correlation IDs   • Multiple Formats                  │    │
│   │ • Auto Rotation     • Error Serialization               │    │
│
└──────────────────────────────────────────────────────────────────────────────────────
│
├──────────────────────────────────────────────────────────────────────────────────────
│                    INFRASTRUCTURE LAYER                             │
│                                                                      │
│ • Configuration Management   • Security & Auth                      │
│ • Service Discovery         • Load Balancing                        │
│ • Health Checks             • Metrics Collection                    │
│ • Message Queues           • Cache Management                       │
```

```
└──────────────────────────────────────────────────────────────────────────────────────────
```

Data Flow Patterns

1. Ingestion → Processing → Analysis

```python
# Example: Document processing pipeline
Text Document → Attention Engine (embeddings) → Knowledge Graph (entities/relations)
```

2. Real-time Analysis

```python
# Example: Live system monitoring
Metrics Stream → Analytics Engine → Knowledge Graph (system relationships) → Real-ti
```

3. Cross-module Queries

```python
# Example: Intelligent search
Query → Knowledge Graph (semantic search) → Attention Engine (relevance) → Analytics
```

Unified Configuration Management

Central Configuration System

python

```python
# config/platform_config.py

from pathlib import Path
from dataclasses import dataclass, field
from typing import Dict, Any, Optional
import os
import yaml

@dataclass
class PlatformConfig:
    """Unified configuration for the entire platform."""

    # Environment
    environment: str = "development"  # development, staging, production
    debug: bool = False

    # Core directories
    base_dir: Path = field(default_factory=lambda: Path("./enterprise_ai_platform"))
    data_dir: Path = field(default_factory=lambda: Path("./data"))
    logs_dir: Path = field(default_factory=lambda: Path("./logs"))
    cache_dir: Path = field(default_factory=lambda: Path("./cache"))

    # Logger configuration
    logger: Dict[str, Any] = field(default_factory=lambda: {
        "min_level": "INFO",
        "enable_console": True,
        "log_file": "./logs/platform.log",
        "format": "structured",
        "enable_batching": True,
        "batch_size": 500,
        "flush_interval": 5.0,
        "rotate_size": 100 * 1024 * 1024,  # 100MB
        "max_files": 10,
        "enable_correlation": True,
        "enable_metrics": True
    })

    # Analytics configuration
    analytics: Dict[str, Any] = field(default_factory=lambda: {
        "output_directory": "./analytics_output",
        "data_retention_days": 90,
        "worker_pool_size": 8,
        "max_memory_usage_mb": 2048,
        "cache_ttl_seconds": 3600,
        "batch_size": 1000
        "enable_streaming": True,
        "enable_performance_monitoring": True,
        "chart_dpi": 300,
        "default_export_format": "JSON"
    })

    # Knowledge Graph configuration
    knowledge_graph: Dict[str, Any] = field(default_factory=lambda: {
        "max_workers": 8,
        "enable_performance_monitoring": True,
        "cache_size": 10000,
        "memory_limit_gb": 4,
        "auto_save_interval": 300,  # 5 minutes
        "backup_retention_days": 30,
        "indexing_strategy": "hash",
        "enable_spatial_index": True,
        "enable_text_index": True,
        "query_cache_size": 5000,
        "max_query_depth": 20
    })

    # Attention Engine configuration
    attention: Dict[str, Any] = field(default_factory=lambda: {
        "hidden_size": 768,
        "num_heads": 12,
        "attention_type": "multi_head",
        "positional_encoding": "rotary",
        "use_flash_attention": True,
        "memory_efficient": True,
        "enable_profiling": True,
        "enable_pattern_analysis": True,
        "max_sequence_length": 2048,
        "attention_dropout": 0.1,
        "use_mixed_precision": True
    })

    # Database configuration
    database: Dict[str, Any] = field(default_factory=lambda: {
        "url": os.getenv("DATABASE_URL", "sqlite:///./platform.db"),
        "pool_size": 10,
        "max_overflow": 20,
        "pool_timeout": 30,
        "pool_recycle": 3600
    })

    # Redis configuration (for caching and message queues)
    redis: Dict[str, Any] = field(default_factory=lambda: {
        "url": os.getenv("REDIS_URL", "redis://localhost:6379/0"),
        "max_connections": 50,
        "socket_timeout": 30,
        "socket_connect_timeout": 30,
        "retry_on_timeout": True
    })

    # API configuration
    api: Dict[str, Any] = field(default_factory=lambda: {
        "host": "0.0.0.0",
        "port": 8080,
        "workers": 4,
        "max_request_size": 100 * 1024 * 1024,  # 100MB
        "timeout": 300,
        "cors_origins": ["*"],
        "rate_limiting": {
            "enabled": True,
            "requests_per_minute": 1000
        }
    })

    # Security configuration
    security: Dict[str, Any] = field(default_factory=lambda: {
        "secret_key": os.getenv("SECRET_KEY", "dev-secret-key-change-in-production"),
        "algorithm": "HS256",
        "access_token_expire_minutes": 30,
        "enable_cors": True,
        "enable_https": False,
        "max_login_attempts": 5,
        "lockout_duration_minutes": 15
    })

    # Monitoring configuration
    monitoring: Dict[str, Any] = field(default_factory=lambda: {
        "enabled": True,
        "metrics_interval": 60,  # seconds
        "health_check_interval": 30,  # seconds
        "alert_thresholds": {
            "cpu_percent": 80,
            "memory_percent": 85,
            "disk_percent": 90,
            "error_rate_percent": 5
        },
        "prometheus_enabled": False,
        "prometheus_port": 9090
    })

    def __post_init__(self):
        """Validate and setup configuration after initialization."""
        # Ensure directories exist
        for dir_path in [self.base_dir, self.data_dir, self.logs_dir, self.cache_dir]:
            dir_path.mkdir(parents=True, exist_ok=True)

        # Update nested configurations with computed paths
        self.logger["log_file"] = str(self.logs_dir / "platform.log")
        self.analytics["output_directory"] = str(self.base_dir / "analytics_output")

        # Environment-specific overrides
        if self.environment == "production":
            self.debug = False
            self.logger["min_level"] = "INFO"
            self.security["enable_https"] = True
        elif self.environment == "development":
            self.debug = True
            self.logger["min_level"] = "DEBUG"
            self.logger["enable_console"] = True

    @classmethod
    def from_file(cls, config_path: Path) -> 'PlatformConfig':
        """Load configuration from YAML file."""
        if not config_path.exists():
            return cls()

        with open(config_path, 'r') as f:
            config_data = yaml.safe_load(f)

        return cls(**config_data)

    def to_file(self, config_path: Path) -> None:
        """Save configuration to YAML file."""
        config_path.parent.mkdir(parents=True, exist_ok=True)

        # Convert to dict for serialization
        import dataclasses
        config_dict = dataclasses.asdict(self)

        # Convert Path objects to strings
        def convert_paths(obj):
            if isinstance(obj, dict):
                return {k: convert_paths(v) for k, v in obj.items()}
            elif isinstance(obj, list):
                return [convert_paths(item) for item in obj]
```

```python
                return [convert_paths(item) for item in obj]
            elif isinstance(obj, Path):
                return str(obj)
            return obj

        config_dict = convert_paths(config_dict)

        with open(config_path, 'w') as f:
            yaml.dump(config_dict, f, default_flow_style=False, indent=2)

    def get_logger_config(self):
        """Get logger-specific configuration."""
        from enterprise_ai_infrastructure.enterprise_logger import LoggerConfig, LogLev

        return LoggerConfig(
            minLevel=LogLevel[self.logger["min_level"]],
            enableConsole=self.logger["enable_console"],
            logFile=self.logger["log_file"],
            format=LogFormat[self.logger["format"].upper()],
            enableBatching=self.logger["enable_batching"],
            batchSize=self.logger["batch_size"],
            flushInterval=self.logger["flush_interval"],
            rotateSize=self.logger["rotate_size"],
            maxFiles=self.logger["max_files"],
            enableCorrelation=self.logger["enable_correlation"],
            enableMetrics=self.logger["enable_metrics"]
        )

    def get_analytics_config(self):
        """Get analytics-specific configuration."""
        from enterprise_ai_infrastructure.enterprise_analytics.core import AnalyticsCor

        return AnalyticsConfig(
            output_directory=Path(self.analytics["output_directory"]),
            data_retention_days=self.analytics["data_retention_days"],
            worker_pool_size=self.analytics["worker_pool_size"],
            max_memory_usage_mb=self.analytics["max_memory_usage_mb"],
            cache_ttl_seconds=self.analytics["cache_ttl_seconds"],
            batch_size=self.analytics["batch_size"],
            enable_streaming=self.analytics["enable_streaming"],
            enable_performance_monitoring=self.analytics["enable_performance_monitoring
            chart_dpi=self.analytics["chart_dpi"],
            default_export_format=ExportFormat[self.analytics["default_export_format"]]
        )

    def get_attention_config(self):
        """Get attention-specific configuration."""
        from enterprise_ai_infrastructure.enterprise_attention import (
            AttentionConfig, AttentionType, PositionalEncoding
        )

        return AttentionConfig(
            hidden_size=self.attention["hidden_size"],
            num_heads=self.attention["num_heads"],
            attention_type=AttentionType[self.attention["attention_type"].upper()],
            positional_encoding=PositionalEncoding[self.attention["positional_encoding'
            use_flash_attention=self.attention["use_flash_attention"],
            memory_efficient=self.attention["memory_efficient"],
            enable_profiling=self.attention["enable_profiling"],
            enable_pattern_analysis=self.attention["enable_pattern_analysis"],
            max_sequence_length=self.attention["max_sequence_length"],
            attention_dropout=self.attention["attention_dropout"],
            use_mixed_precision=self.attention["use_mixed_precision"]
        )


# Global configuration instance
_config = None

def get_config() -> PlatformConfig:
    """Get the global platform configuration."""
    global _config
    if _config is None:
        config_path = Path("config/platform.yaml")
        _config = PlatformConfig.from_file(config_path)
    return _config

def set_config(config: PlatformConfig) -> None:
    """Set the global platform configuration."""
    global _config
    _config = config
```

Environment-Specific Configurations

yaml

```yaml
# config/environments/development.yaml
environment: development
debug: true

logger:
  min_level: DEBUG
  enable_console: true
  format: text

analytics:
  worker_pool_size: 4
  max_memory_usage_mb: 1024

attention:
  enable_profiling: true
  enable_pattern_analysis: true

monitoring:
  metrics_interval: 30
  health_check_interval: 15
```

yaml

```yaml
# config/environments/production.yaml
environment: production
debug: false

logger:
  min_level: INFO
  enable_console: false
  format: structured
  batch_size: 1000
  flush_interval: 2.0

analytics:
  worker_pool_size: 16
  max_memory_usage_mb: 8192
  enable_streaming: true

security:
  enable_https: true
  max_login_attempts: 3

monitoring:
  enabled: true
  prometheus_enabled: true
  alert_thresholds:
    cpu_percent: 70
    memory_percent: 80
    disk_percent: 85
```

Module Integration Patterns

1. Shared Context Pattern

python

```python
# platform/context.py

import asyncio
from typing import Optional, Dict, Any
from contextlib import asynccontextmanager

from enterprise_ai_infrastructure.enterprise_logger import Logger
from enterprise_ai_infrastructure.enterprise_analytics.core import AnalyticsEngine
from enterprise_ai_infrastructure.enterprise_knowledge_graph import KnowledgeGraph
from enterprise_ai_infrastructure.enterprise_attention import AttentionFactory

class PlatformContext:
    """Shared context for all platform modules."""

    def __init__(self, config: PlatformConfig):
        self.config = config
        self.correlation_id = None

        # Initialize core components
        self.logger = Logger("Platform", config.get_logger_config())
        self.analytics = AnalyticsEngine(config.get_analytics_config())
        self.knowledge_graph = KnowledgeGraph(config.knowledge_graph)

        # Attention engine (framework-specific)
        self.attention_pytorch = AttentionFactory.create_attention(
            config.get_attention_config(), "pytorch"
        )

        # Shared state
        self._session_data: Dict[str, Any] = {}
        self._performance_metrics: Dict[str, list] = {}

    async def initialize(self):
        """Initialize all components asynchronously."""
        self.logger.info("Initializing platform context")

        # Setup analytics data sources
        await self._setup_analytics_sources()

        # Initialize knowledge graph schema
        await self._setup_knowledge_graph_schema()

        self.logger.info("Platform context initialized successfully")

    async def _setup_analytics_sources(self):
```

```python
async def _setup_analytics_sources(self):
    """Setup analytics data sources."""
    # Register platform logger as a data source
    class LoggerDataSource:
        def __init__(self, logger):
            self.logger = logger

        async def get_metrics(self, start_time, end_time, metric_names=None):
            metrics = self.logger.get_metrics()
            # Convert logger metrics to analytics data points
            from enterprise_ai_infrastructure.enterprise_analytics.core import Base
            import datetime

            points = []
            timestamp = datetime.datetime.now()

            for metric_name, value in metrics.__dict__.items():
                if isinstance(value, (int, float)):
                    points.append(BaseDataPoint(
                        timestamp=timestamp,
                        value=value,
                        source="platform_logger",
                        metadata={"metric_type": metric_name}
                    ))

            return points

        async def get_health_status(self):
            return {"status": "healthy", "component": "logger"}

    await self.analytics.register_data_source("platform_logger", LoggerDataSource(

async def _setup_knowledge_graph_schema(self):
    """Setup knowledge graph schemas."""
    from enterprise_ai_infrastructure.enterprise_knowledge_graph import NodeType, 

    # Define platform-specific schemas
    self.knowledge_graph.schema.define_node_schema(
        NodeType.ENTITY,
        required_properties=["name", "type"],
        property_types={"name": str, "type": str, "confidence": float}
    )

    self.knowledge_graph.schema.define_edge_schema(
        EdgeType.RELATED_TO,
        required_properties=["relationship_type"],
        property_types={"relationship_type": str, "strength": float}
    )

async def close(self):
    """Clean up all resources."""
    self.logger.info("Shutting down platform context")

    await self.analytics.close()
    await self.knowledge_graph.close()
    await self.logger.close()

@asynccontextmanager
async def correlation_context(self, correlation_id: str = None):
    """Context manager for correlation tracking."""
    if correlation_id is None:
        correlation_id = f"ctx-{int(time.time() * 1000)}-{uuid.uuid4().hex[:8]}"

    old_correlation_id = self.correlation_id
    self.correlation_id = correlation_id

    self.logger.info(f"Starting correlated operation", correlation_id=correlation_

    try:
        yield correlation_id
    finally:
        self.logger.info(f"Completed correlated operation", correlation_id=correlat
        self.correlation_id = old_correlation_id

def log(self, level: str, message: str, **kwargs):
    """Centralized logging with correlation."""
    log_method = getattr(self.logger, level.lower())
    log_method(message, correlation_id=self.correlation_id, **kwargs)
```

2. Event-Driven Integration

python

```python
# platform/events.py

import asyncio
from typing import Dict, List, Callable, Any
from dataclasses import dataclass
from enum import Enum

class EventType(Enum):
    """Platform event types."""
    NODE_CREATED = "node_created"
    NODE_UPDATED = "node_updated"
    NODE_DELETED = "node_deleted"
    EDGE_CREATED = "edge_created"
    ANALYSIS_COMPLETED = "analysis_completed"
    ATTENTION_COMPUTED = "attention_computed"
    REPORT_GENERATED = "report_generated"
    ERROR_OCCURRED = "error_occurred"

@dataclass
class PlatformEvent:
    """Platform event structure."""
    type: EventType
    source: str
    data: Dict[str, Any]
    correlation_id: Optional[str] = None
    timestamp: datetime.datetime = field(default_factory=datetime.datetime.now)

class EventBus:
    """Simple event bus for module communication."""

    def __init__(self):
        self._handlers: Dict[EventType, List[Callable]] = {}
        self._event_queue = asyncio.Queue()
        self._running = False
        self._processor_task = None

    def subscribe(self, event_type: EventType, handler: Callable):
        """Subscribe to events of a specific type."""
        if event_type not in self._handlers:
            self._handlers[event_type] = []
        self._handlers[event_type].append(handler)

    async def publish(self, event: PlatformEvent):
        """Publish an event to the bus."""
        await self._event_queue.put(event)
        await self._event_queue.put(event)

    async def start(self):
        """Start the event processor."""
        self._running = True
        self._processor_task = asyncio.create_task(self._process_events())

    async def stop(self):
        """Stop the event processor."""
        self._running = False
        if self._processor_task:
            self._processor_task.cancel()
            try:
                await self._processor_task
            except asyncio.CancelledError:
                pass

    async def _process_events(self):
        """Process events from the queue."""
        while self._running:
            try:
                event = await asyncio.wait_for(self._event_queue.get(), timeout=1.0)
                await self._handle_event(event)
            except asyncio.TimeoutError:
                continue
            except Exception as e:
                print(f"Error processing event: {e}")

    async def _handle_event(self, event: PlatformEvent):
        """Handle a single event."""
        handlers = self._handlers.get(event.type, [])

        if handlers:
            tasks = [handler(event) for handler in handlers]
            await asyncio.gather(*tasks, return_exceptions=True)

# Example event handlers
class IntegratedEventHandlers:
    """Event handlers that coordinate between modules."""

    def __init__(self, context: PlatformContext):
        self.context = context

    async def on_node_created(self, event: PlatformEvent):
        """Handle node creation events."""
        node_data = event.data
        # Log the event
        self.context.log("info", f"Node created: {node_data.get('id')}",
                        node_type=node_data.get('type'),
                        correlation_id=event.correlation_id)

        # Update analytics
        from enterprise_ai_infrastructure.enterprise_analytics.core import BaseDataPoi
        data_point = BaseDataPoint(
            timestamp=event.timestamp,
            value=1,
            source="knowledge_graph",
            metadata={
                "event_type": "node_created",
                "node_type": node_data.get('type'),
                "node_id": node_data.get('id')
            }
        )

        # Store analytics point (would integrate with analytics pipeline)
        self.context.log("debug", "Analytics updated for node creation")

    async def on_analysis_completed(self, event: PlatformEvent):
        """Handle analysis completion events."""
        analysis_data = event.data

        # Generate report if needed
        if analysis_data.get('generate_report', False):
            self.context.log("info", "Generating report for completed analysis",
                            analysis_type=analysis_data.get('type'),
                            correlation_id=event.correlation_id)

            # Trigger report generation (would integrate with reporting engine)
            # await self.context.analytics.generate_report(...)

    async def on_error_occurred(self, event: PlatformEvent):
        """Handle error events."""
        error_data = event.data

        # Log with full context
        self.context.log("error", f"Platform error: {error_data.get('message')}",
                        component=event.source,
                        error_type=error_data.get('type'),
                        correlation_id=event.correlation_id)

        # Could trigger alerts, notifications, etc.
```

3. Pipeline Integration Pattern

3. Pipeline Integration Pattern

python

```python
# platform/pipeline.py

import asyncio
from typing import List, Any, Dict, Optional, Callable
from abc import ABC, abstractmethod

class PipelineStage(ABC):
    """Abstract base class for pipeline stages."""

    def __init__(self, name: str, context: PlatformContext):
        self.name = name
        self.context = context

    @abstractmethod
    async def process(self, data: Any, metadata: Dict[str, Any]) -> Any:
        """Process data through this stage."""
        pass

    async def _log_stage_start(self, metadata: Dict[str, Any]):
        """Log stage start."""
        self.context.log("debug", f"Pipeline stage '{self.name}' starting",
                         stage=self.name,
                         correlation_id=metadata.get('correlation_id'))

    async def _log_stage_complete(self, metadata: Dict[str, Any], duration: float):
        """Log stage completion."""
        self.context.log("debug", f"Pipeline stage '{self.name}' completed",
                         stage=self.name,
                         duration_ms=duration * 1000,
                         correlation_id=metadata.get('correlation_id'))

class TextProcessingStage(PipelineStage):
    """Stage for processing text through attention mechanisms."""

    async def process(self, data: str, metadata: Dict[str, Any]) -> Dict[str, Any]:
        import time
        start_time = time.time()
        await self._log_stage_start(metadata)

        try:
            # Tokenize and prepare input
            import torch

            # Simple tokenization (in practice, use proper tokenizer)
            tokens = data.split()
            tokens = data.split()
            vocab_size = 1000  # Simplified
            token_ids = [hash(token) % vocab_size for token in tokens]

            # Create input tensor
            max_length = self.context.config.attention["max_sequence_length"]
            if len(token_ids) > max_length:
                token_ids = token_ids[:max_length]

            # Pad to fixed length for demo
            while len(token_ids) < min(len(tokens) + 10, max_length):
                token_ids.append(0)  # Padding token

            input_tensor = torch.randn(1, len(token_ids),
                                     self.context.config.attention["hidden_size"])

            # Process through attention
            result = self.context.attention_pytorch.forward(input_tensor)

            # Extract embeddings (last layer)
            embeddings = result['output'].mean(dim=1).squeeze(0).detach().numpy()

            processed_data = {
                'original_text': data,
                'embeddings': embeddings,
                'attention_weights': result.get('attention_weights'),
                'token_count': len(tokens)
            }

            await self._log_stage_complete(metadata, time.time() - start_time)
            return processed_data

        except Exception as e:
            self.context.log("error", f"Text processing failed: {str(e)}",
                           stage=self.name, correlation_id=metadata.get('correlation_ic
            raise

class KnowledgeExtractionStage(PipelineStage):
    """Stage for extracting knowledge and updating graph."""

    async def process(self, data: Dict[str, Any], metadata: Dict[str, Any]) -> Dict[st
        import time
        start_time = time.time()
        await self._log_stage_start(metadata)

        try:
            # Extract entities and relationships from processed text
```

```python
text = data['original_text']
embeddings = data['embeddings']

# Simple entity extraction (in practice, use NER)
words = text.split()
entities = [word for word in words if word[0].isupper()]  # Simplified

# Create nodes in knowledge graph
from enterprise_ai_infrastructure.enterprise_knowledge_graph import (
    GraphNode, GraphEdge, NodeType, EdgeType
)

nodes_created = []
edges_created = []

for i, entity in enumerate(entities):
    node_id = f"entity_{hash(entity) % 10000}"

    # Check if node already exists
    existing_node = await self.context.knowledge_graph.get_node(node_id)
    if not existing_node:
        node = GraphNode(
            id=node_id,
            type=NodeType.ENTITY,
            label=entity,
            properties={
                'source_text': text,
                'confidence': 0.8,
                'extraction_method': 'simple_capitalization'
            },
            embedding=embeddings if i == 0 else None  # Only first entity
        )

        await self.context.knowledge_graph.add_node(node)
        nodes_created.append(node_id)

# Create relationships between consecutive entities
for i in range(len(entities) - 1):
    source_id = f"entity_{hash(entities[i]) % 10000}"
    target_id = f"entity_{hash(entities[i + 1]) % 10000}"
    edge_id = f"edge_{hash(f'{source_id}_{target_id}') % 10000}"

    edge = GraphEdge(
        id=edge_id,
        source_id=source_id,
        target_id=target_id,
                    type=EdgeType.RELATED_TO,
                    label="co_occurs",
                    properties={
                        'relationship_type': 'co_occurrence',
                        'strength': 0.7,
                        'source_text': text
                    }
                )

                try:
                    await self.context.knowledge_graph.add_edge(edge)
                    edges_created.append(edge_id)
                except Exception as e:
                    # Edge might already exist or nodes might not exist
                    self.context.log("debug", f"Could not create edge: {str(e)}")

            result = {
                **data,
                'entities_extracted': entities,
                'nodes_created': nodes_created,
                'edges_created': edges_created,
                'extraction_stats': {
                    'entity_count': len(entities),
                    'node_count': len(nodes_created),
                    'edge_count': len(edges_created)
                }
            }

            await self._log_stage_complete(metadata, time.time() - start_time)
            return result

        except Exception as e:
            self.context.log("error", f"Knowledge extraction failed: {str(e)}",
                           stage=self.name, correlation_id=metadata.get('correlation_ic

class AnalyticsStage(PipelineStage):
    """Stage for generating analytics and insights."""

    async def process(self, data: Dict[str, Any], metadata: Dict[str, Any]) -> Dict[str
        import time
        start_time = time.time()
        await self._log_stage_start(metadata)

        try:
            # Create analytics data points
            from enterprise_ai_infrastructure.enterprise_analytics.core import BaseDat
```

```python
import datetime

timestamp = datetime.datetime.now()

data_points = []

# Entity extraction metrics
if 'extraction_stats' in data:
    stats = data['extraction_stats']

    for metric_name, value in stats.items():
        data_points.append(BaseDataPoint(
            timestamp=timestamp,
            value=value,
            source="knowledge_extraction",
            metadata={
                'metric_type': metric_name,
                'text_length': len(data.get('original_text', '')),
                'pipeline_stage': 'knowledge_extraction'
            }
        ))

# Token processing metrics
if 'token_count' in data:
    data_points.append(BaseDataPoint(
        timestamp=timestamp,
        value=data['token_count'],
        source="text_processing",
        metadata={
            'metric_type': 'token_count',
            'pipeline_stage': 'text_processing'
        }
    ))

# Generate insights
insights = await self._generate_insights(data)

result = {
    **data,
    'analytics_data_points': data_points,
    'insights': insights,
    'analytics_timestamp': timestamp.isoformat()
}

await self._log_stage_complete(metadata, time.time() - start_time)
return result
```

```
        except Exception as e:
            self.context.log("error", f"Analytics processing failed: {str(e)}",
                           stage=self.name, correlation_id=metadata.get('correlation_ic
            raise

    async def _generate_insights(self, data: Dict[str, Any]) -> Dict[str, Any]:
        """Generate insights from processed data."""
        insights = {}

        # Text complexity
        text = data.get('original_text', '')
        if text:
            words = text.split()
            avg_word_length = sum(len(word) for word in words) / len(words) if words el
            insights['text_complexity'] = {
                'word_count': len(words),
                'avg_word_length': avg_word_length,
                'complexity_score': min(avg_word_length / 5.0, 1.0)  # Normalized
            }

        # Entity extraction quality
        if 'extraction_stats' in data:
            stats = data['extraction_stats']
            entity_density = stats['entity_count'] / len(text.split()) if text else 0
            insights['extraction_quality'] = {
                'entity_density': entity_density,
                'extraction_efficiency': stats['node_count'] / max(stats['entity_count'
            }

        return insights

class IntegratedPipeline:
    """Integrated processing pipeline."""

    def __init__(self, context: PlatformContext):
        self.context = context
        self.stages: List[PipelineStage] = []

    def add_stage(self, stage: PipelineStage):
        """Add a stage to the pipeline."""
        self.stages.append(stage)

    async def process(self, initial_data: Any, correlation_id: str = None) -> Dict[str,
        """Process data through all stages."""
        if correlation_id is None:
            import uuid
            import uuid
            correlation_id = f"pipeline-{uuid.uuid4().hex[:8]}"

        metadata = {'correlation_id': correlation_id}

        async with self.context.correlation_context(correlation_id):
            self.context.log("info", "Starting integrated pipeline processing",
                           stage_count=len(self.stages))

            data = initial_data
            stage_results = []

            try:
                for i, stage in enumerate(self.stages):
                    self.context.log("debug", f"Processing stage {i+1}/{len(self.stage

                    stage_start = time.time()
                    data = await stage.process(data, metadata)
                    stage_duration = time.time() - stage_start

                    stage_results.append({
                        'stage': stage.name,
                        'duration_ms': stage_duration * 1000,
                        'output_size': len(str(data)) if isinstance(data, (str, dict))
                    })

                total_duration = sum(r['duration_ms'] for r in stage_results) / 1000

                self.context.log("info", "Pipeline processing completed successfully",
                               total_duration_ms=total_duration * 1000,
                               stages_completed=len(stage_results))

                return {
                    'result': data,
                    'pipeline_metadata': {
                        'correlation_id': correlation_id,
                        'total_duration_ms': total_duration * 1000,
                        'stage_results': stage_results,
                        'timestamp': datetime.datetime.now().isoformat()
                    }
                }

            except Exception as e:
                self.context.log("error", f"Pipeline processing failed: {str(e)}",
                               completed_stages=len(stage_results),
                               failed_at_stage=len(stage_results) + 1 if len(stage_resu
                raise
```

End-to-End Examples

Example 1: Document Intelligence Pipeline

python

```python
# examples/document_intelligence.py

import asyncio
from pathlib import Path

async def document_intelligence_example():
    """Complete document processing pipeline example."""

    # Initialize platform
    from platform.context import PlatformContext
    from platform.pipeline import IntegratedPipeline, TextProcessingStage, KnowledgeExt
    from config.platform_config import PlatformConfig

    # Load configuration
    config = PlatformConfig.from_file(Path("config/platform.yaml"))

    # Create platform context
    context = PlatformContext(config)
    await context.initialize()

    try:
        # Create integrated pipeline
        pipeline = IntegratedPipeline(context)
        pipeline.add_stage(TextProcessingStage("text_processing", context))
        pipeline.add_stage(KnowledgeExtractionStage("knowledge_extraction", context))
        pipeline.add_stage(AnalyticsStage("analytics", context))

        # Sample document
        document_text = """
        Apple Inc. is a technology company founded by Steve Jobs and Steve Wozniak.
        The company is headquartered in Cupertino, California.
        Apple develops innovative products like the iPhone, iPad, and MacBook.
        Tim Cook is the current CEO of Apple Inc.
        """

        # Process document
        context.logger.info("Starting document intelligence pipeline")

        result = await pipeline.process(document_text)

        # Extract results
        final_data = result['result']
        pipeline_metadata = result['pipeline_metadata']

        # Display results
        # Display results
        print("\n=== DOCUMENT INTELLIGENCE RESULTS ===")
        print(f"Correlation ID: {pipeline_metadata['correlation_id']}")
        print(f"Total Processing Time: {pipeline_metadata['total_duration_ms']:.2f}ms")

        print(f"\nEntities Extracted: {final_data['entities_extracted']}")
        print(f"Nodes Created: {len(final_data['nodes_created'])}")
        print(f"Edges Created: {len(final_data['edges_created'])}")

        print(f"\nInsights Generated:")
        for insight_type, insight_data in final_data['insights'].items():
            print(f"  {insight_type}: {insight_data}")

        # Generate report
        await generate_document_report(context, final_data, pipeline_metadata)

        # Query knowledge graph
        await query_extracted_knowledge(context, final_data)

    finally:
        await context.close()

async def generate_document_report(context, data, metadata):
    """Generate a comprehensive report."""
    from enterprise_ai_infrastructure.enterprise_analytics.reporting import (
        ReportingEngine, ReportConfig, ReportFormat, ReportTemplate
    )

    # Convert data to analytics format
    from enterprise_ai_infrastructure.enterprise_analytics.core import BaseDataPoint
    import datetime

    # Create data points for reporting
    data_points = []
    timestamp = datetime.datetime.now()

    # Add extraction metrics
    if 'extraction_stats' in data:
        for metric, value in data['extraction_stats'].items():
            data_points.append(BaseDataPoint(
                timestamp=timestamp,
                value=value,
                source="document_processing",
                metadata={'metric_type': metric, 'document_id': metadata['correlation_
            ))

    # Create reporting engine
```

```python
    reporting_engine = ReportingEngine(context.analytics.config)

    # Configure report
    report_config = ReportConfig(
        title="Document Intelligence Report",
        subtitle=f"Analysis Results - {timestamp.strftime('%Y-%m-%d %H:%M:%S')}",
        author="Enterprise AI Platform",
        template=ReportTemplate.TECHNICAL_ANALYSIS,
        output_format=ReportFormat.HTML
    )

    # Generate report
    report_path = await reporting_engine.generate_comprehensive_report(
        data_points, report_config, context.analytics.viz_engine
    )

    context.logger.info(f"Document report generated: {report_path}")
    print(f"\nReport generated: {report_path}")

async def query_extracted_knowledge(context, data):
    """Query the knowledge graph for insights."""
    print(f"\n=== KNOWLEDGE GRAPH QUERIES ===")

    # Find all entities
    all_entities = await context.knowledge_graph.search_nodes({
        "type": "entity"
    })
    print(f"Total entities in graph: {len(all_entities)}")

    # Find relationships
    if data['nodes_created']:
        first_node_id = data['nodes_created'][0]
        neighbors = await context.knowledge_graph.get_neighbors(first_node_id)
        print(f"Neighbors of {first_node_id}: {neighbors}")

    # Calculate centrality
    if len(all_entities) > 1:
        centrality = await context.knowledge_graph.calculate_centrality("degree")
        most_central = max(centrality.items(), key=lambda x: x[1])
        print(f"Most central entity: {most_central[0]} (centrality: {most_central[1]:.

if __name__ == "__main__":
    asyncio.run(document_intelligence_example())
```

Example 2: Real-time Analytics Dashboard

python

```python
# examples/realtime_dashboard.py

import asyncio
import random
from datetime import datetime, timedelta

async def realtime_dashboard_example():
    """Real-time analytics dashboard with live updates."""

    from platform.context import PlatformContext
    from config.platform_config import PlatformConfig
    from pathlib import Path

    # Load configuration optimized for real-time processing
    config = PlatformConfig.from_file(Path("config/platform.yaml"))
    config.analytics["enable_streaming"] = True
    config.analytics["batch_size"] = 100
    config.logger["flush_interval"] = 1.0  # Faster flushing for real-time

    context = PlatformContext(config)
    await context.initialize()

    try:
        # Setup real-time data sources
        await setup_realtime_sources(context)

        # Start dashboard
        await run_dashboard(context)

    finally:
        await context.close()

async def setup_realtime_sources(context):
    """Setup real-time data sources."""

    class SimulatedMetricsSource:
        """Simulates real-time system metrics."""

        def __init__(self):
            self.start_time = datetime.now()

        async def get_metrics(self, start_time, end_time, metric_names=None):
            from enterprise_ai_infrastructure.enterprise_analytics.core import BaseDat

            # Simulate system metrics
            # Simulate system metrics
            current_time = datetime.now()

            metrics = []

            # CPU usage (trending upward during business hours)
            hour = current_time.hour
            base_cpu = 30 + (hour - 9) * 2 if 9 <= hour <= 17 else 20
            cpu_usage = max(0, min(100, base_cpu + random.uniform(-10, 15)))

            metrics.append(BaseDataPoint(
                timestamp=current_time,
                value=cpu_usage,
                source="system_monitor",
                metadata={"metric_type": "cpu_usage", "unit": "percent"}
            ))

            # Memory usage
            memory_usage = random.uniform(40, 80)
            metrics.append(BaseDataPoint(
                timestamp=current_time,
                value=memory_usage,
                source="system_monitor",
                metadata={"metric_type": "memory_usage", "unit": "percent"}
            ))

            # Request rate
            request_rate = random.uniform(50, 200)
            metrics.append(BaseDataPoint(
                timestamp=current_time,
                value=request_rate,
                source="application",
                metadata={"metric_type": "request_rate", "unit": "requests_per_minute"}
            ))

            # Error rate
            error_rate = random.uniform(0, 5)
            metrics.append(BaseDataPoint(
                timestamp=current_time,
                value=error_rate,
                source="application",
                metadata={"metric_type": "error_rate", "unit": "errors_per_minute"}
            ))

            return metrics

        async def get_health_status(self):
```

```python
            return {"status": "healthy", "component": "metrics_simulator"}

    # Register the simulated source
    await context.analytics.register_data_source("realtime_metrics", SimulatedMetricsSc

    context.logger.info("Real-time data sources configured")

async def run_dashboard(context):
    """Run the real-time dashboard."""

    print("\n=== REAL-TIME ANALYTICS DASHBOARD ===")
    print("Press Ctrl+C to stop...")

    # Storage for dashboard data
    metrics_history = []
    max_history = 100  # Keep last 100 data points

    update_interval = 5  # Update every 5 seconds

    try:
        while True:
            # Collect current metrics
            end_time = datetime.now()
            start_time = end_time - timedelta(seconds=update_interval)

            metrics = await context.analytics.collect_metrics(
                "realtime_metrics", start_time, end_time
            )

            # Add to history
            metrics_history.extend(metrics)

            # Trim history
            if len(metrics_history) > max_history:
                metrics_history = metrics_history[-max_history:]

            # Update dashboard display
            await update_dashboard_display(context, metrics, metrics_history)

            # Check for anomalies and alerts
            await check_alerts(context, metrics)

            # Wait for next update
            await asyncio.sleep(update_interval)

    except KeyboardInterrupt:
        print("\nStopping dashboard...")

async def update_dashboard_display(context, current_metrics, history):
    """Update the dashboard display."""

    # Clear screen (simple approach)
    import os
    os.system('clear' if os.name == 'posix' else 'cls')

    print("=== REAL-TIME ANALYTICS DASHBOARD ===")
    print(f"Last Update: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"Data Points in History: {len(history)}")

    # Current metrics
    print(f"\n--- CURRENT METRICS ---")
    metric_types = {}
    for metric in current_metrics:
        metric_type = metric.metadata.get('metric_type', 'unknown')
        metric_types[metric_type] = metric.value

    for metric_type, value in metric_types.items():
        # Status indicators based on thresholds
        if metric_type in ['cpu_usage', 'memory_usage']:
            status = "CRITICAL" if value > 80 else "WARNING" if value > 60 else "NORMAl
        elif metric_type == 'error_rate':
            status = "CRITICAL" if value > 3 else "WARNING" if value > 1 else "NORMAL"
        else:
            status = "NORMAL"

        print(f"[{status}] {metric_type}: {value:.1f}")

    # Trends (simple calculation)
    if len(history) >= 10:
        print(f"\n--- TRENDS (Last 10 data points) ---")

        # Group by metric type
        metric_trends = {}
        recent_history = history[-10:]

        for metric in recent_history:
            metric_type = metric.metadata.get('metric_type', 'unknown')
            if metric_type not in metric_trends:
                metric_trends[metric_type] = []
            metric_trends[metric_type].append(metric.value)

        for metric_type, values in metric_trends.items():
            if len(values) >= 2:
```

```
            trend = "INCREASING" if values[-1] > values[0] else "DECREASING" if va
            avg_change = (values[-1] - values[0]) / len(values)
            print(f"[{trend}] {metric_type}: {avg_change:+.1f} average change")

    # System status
    print(f"\n--- SYSTEM STATUS ---")

    # Calculate overall health score
    health_score = 100
    for metric_type, value in metric_types.items():
        if metric_type in ['cpu_usage', 'memory_usage'] and value > 80:
            health_score -= 20
        elif metric_type == 'error_rate' and value > 3:
            health_score -= 30

    health_status = "HEALTHY" if health_score >= 80 else "WARNING" if health_score >= 
    print(f"Overall Health: {health_status} ({health_score}%)")

async def check_alerts(context, metrics):
    """Check for alert conditions."""

    alerts = []

    for metric in metrics:
        metric_type = metric.metadata.get('metric_type')
        value = metric.value

        # Define alert thresholds
        if metric_type == 'cpu_usage' and value > 85:
            alerts.append(f"High CPU usage: {value:.1f}%")
        elif metric_type == 'memory_usage' and value > 90:
            alerts.append(f"High memory usage: {value:.1f}%")
        elif metric_type == 'error_rate' and value > 5:
            alerts.append(f"High error rate: {value:.1f} errors/min")

    # Log alerts
    for alert in alerts:
        context.logger.warn(f"ALERT: {alert}",
                          alert_type="threshold_exceeded",
                          severity="high")

        # In production, would send to alerting system
        print(f"\nALERT: {alert}")

if __name__ == "__main__":
    asyncio.run(realtime_dashboard_example())
```

Example 3: Semantic Search System

python

```python
# examples/semantic_search.py

import asyncio
import numpy as np
from typing import List, Dict, Any

async def semantic_search_example():
    """Semantic search system using all platform components."""

    from platform.context import PlatformContext
    from config.platform_config import PlatformConfig
    from pathlib import Path

    config = PlatformConfig.from_file(Path("config/platform.yaml"))
    context = PlatformContext(config)
    await context.initialize()

    try:
        # Index documents
        await index_documents(context)

        # Perform semantic searches
        await run_search_examples(context)

        # Generate search analytics
        await generate_search_analytics(context)

    finally:
        await context.close()

async def index_documents(context):
    """Index a collection of documents."""

    documents = [
        {
            "id": "doc_001",
            "title": "Machine Learning Fundamentals",
            "content": "Machine learning is a subset of artificial intelligence that fo
            "category": "technology",
            "author": "Dr. Smith"
        },
        {
            "id": "doc_002",
            "title": "The Future of Transportation",
            "content": "Electric vehicles and autonomous driving are revolutionizing t
            "content": "Electric vehicles and autonomous driving are revolutionizing tr
            "category": "automotive",
            "author": "Jane Doe"
        },
        {
            "id": "doc_003",
            "title": "Renewable Energy Solutions",
            "content": "Solar panels and wind turbines are becoming more efficient and
            "category": "energy",
            "author": "Prof. Johnson"
        },
        {
            "id": "doc_004",
            "title": "Advanced Neural Networks",
            "content": "Transformer architectures and attention mechanisms have revolut
            "category": "technology",
            "author": "Dr. Smith"
        },
        {
            "id": "doc_005",
            "title": "Sustainable Technology",
            "content": "Green technology combines innovation with environmental respons
            "category": "environment",
            "author": "Emily Chen"
        }
    ]

    context.logger.info("Starting document indexing", document_count=len(documents))

    for doc in documents:
        await index_single_document(context, doc)

    context.logger.info("Document indexing completed")

async def index_single_document(context, document: Dict[str, Any]):
    """Index a single document with embeddings and knowledge extraction."""

    # Generate embeddings using attention engine
    full_text = f"{document['title']} {document['content']}"
    embeddings = await generate_text_embeddings(context, full_text)

    # Create document node in knowledge graph
    from enterprise_ai_infrastructure.enterprise_knowledge_graph import (
        GraphNode, GraphEdge, NodeType, EdgeType
    )

    doc_node = GraphNode(
        id=document['id'],
        type=NodeType.DOCUMENT,
        label=document['title'],
        properties={
            'content': document['content'],
            'category': document['category'],
            'author': document['author'],
            'word_count': len(full_text.split()),
            'indexed_at': datetime.now().isoformat()
        },
        embedding=embeddings
    )

    await context.knowledge_graph.add_node(doc_node)

    # Extract and link entities
    entities = extract_simple_entities(document['content'])

    for entity in entities:
        entity_id = f"entity_{hash(entity.lower()) % 10000}"

        # Create or update entity node
        existing_entity = await context.knowledge_graph.get_node(entity_id)
        if not existing_entity:
            entity_node = GraphNode(
                id=entity_id,
                type=NodeType.ENTITY,
                label=entity,
                properties={
                    'entity_type': 'extracted_term',
                    'first_seen': datetime.now().isoformat(),
                    'document_count': 1
                }
            )
            await context.knowledge_graph.add_node(entity_node)
        else:
            # Update document count
            updated_props = existing_entity.properties.copy()
            updated_props['document_count'] = updated_props.get('document_count', 0) +

            updated_entity = GraphNode(
                id=existing_entity.id,
                type=existing_entity.type,
                label=existing_entity.label,
                properties=updated_props,
                embedding=existing_entity.embedding,
                created_at=existing_entity.created_at,
                updated_at=datetime.now(),
                version=existing_entity.version + 1
            )
            await context.knowledge_graph.update_node(updated_entity)

        # Create document-entity relationship
        edge_id = f"doc_entity_{document['id']}_{entity_id}"
        edge = GraphEdge(
            id=edge_id,
            source_id=document['id'],
            target_id=entity_id,
            type=EdgeType.HAS_A,
            label="contains_entity",
            properties={
                'extraction_method': 'simple_keywords',
                'confidence': 0.8
            }
        )

        try:
            await context.knowledge_graph.add_edge(edge)
        except:
            pass  # Edge might already exist

    context.logger.debug(f"Indexed document: {document['id']}",
                        entities_extracted=len(entities),
                        embedding_size=len(embeddings) if embeddings is not None else (

async def generate_text_embeddings(context, text: str) -> np.ndarray:
    """Generate embeddings for text using attention engine."""

    import torch

    # Simplified tokenization and embedding generation
    words = text.lower().split()

    # Create dummy input (in practice, use proper tokenizer)
    vocab_size = 10000
    max_length = min(len(words), context.config.attention["max_sequence_length"])

    # Map words to token IDs
    token_ids = [hash(word) % vocab_size for word in words[:max_length]]

    # Pad to consistent length
    while len(token_ids) < max_length:
        token_ids.append(0)
```

```python
    # Create input tensor
    hidden_size = context.config.attention["hidden_size"]
    input_tensor = torch.randn(1, len(token_ids), hidden_size)

    # Process through attention
    try:
        result = context.attention_pytorch.forward(input_tensor)

        # Extract sentence embedding (mean pooling)
        embeddings = result['output'].mean(dim=1).squeeze(0).detach().numpy()

        return embeddings

    except Exception as e:
        context.logger.warn(f"Could not generate embeddings: {str(e)}")
        return None

def extract_simple_entities(text: str) -> List[str]:
    """Simple entity extraction (in practice, use proper NER)."""

    # Common technical terms and proper nouns
    important_terms = [
        'machine learning', 'artificial intelligence', 'neural networks',
        'tesla', 'apple', 'electric vehicles', 'autonomous driving',
        'solar panels', 'wind turbines', 'renewable energy',
        'transformer', 'attention', 'gpt', 'bert',
        'green technology', 'smart grids'
    ]

    text_lower = text.lower()
    entities = []

    for term in important_terms:
        if term in text_lower:
            entities.append(term)

    # Add capitalized words (likely proper nouns)
    words = text.split()
    for word in words:
        if word[0].isupper() and len(word) > 3 and word not in entities:
            entities.append(word)

    return entities

async def run_search_examples(context):
    """Run example semantic searches."""

    print("\n=== SEMANTIC SEARCH EXAMPLES ===")

    search_queries = [
        "artificial intelligence and machine learning",
        "electric cars and sustainable transport",
        "renewable energy technologies",
        "neural network architectures"
    ]

    for query in search_queries:
        print(f"\n--- Searching for: '{query}' ---")

        results = await semantic_search(context, query, top_k=3)

        for i, result in enumerate(results, 1):
            print(f"{i}. {result['title']} (Score: {result['similarity']:.3f})")
            print(f"   Author: {result['author']} | Category: {result['category']}")
            print(f"   Content: {result['content'][:100]}...")

async def semantic_search(context, query: str, top_k: int = 5) -> List[Dict[str, Any]]:
    """Perform semantic search using embeddings."""

    # Generate query embeddings
    query_embeddings = await generate_text_embeddings(context, query)

    if query_embeddings is None:
        context.logger.warn("Could not generate query embeddings")
        return []

    # Get all document nodes with embeddings
    doc_nodes = await context.knowledge_graph.search_nodes({
        "type": "document"
    })

    # Calculate similarities
    similarities = []

    for doc_node in doc_nodes:
        if doc_node.embedding is not None:
            # Calculate cosine similarity
            doc_embedding = doc_node.embedding

            dot_product = np.dot(query_embeddings, doc_embedding)
            query_norm = np.linalg.norm(query_embeddings)
            doc_norm = np.linalg.norm(doc_embedding)
            doc_norm = np.linalg.norm(doc_embedding)

            if query_norm > 0 and doc_norm > 0:
                similarity = dot_product / (query_norm * doc_norm)
            else:
                similarity = 0.0

            similarities.append({
                'document_id': doc_node.id,
                'title': doc_node.label,
                'content': doc_node.properties.get('content', ''),
                'category': doc_node.properties.get('category', ''),
                'author': doc_node.properties.get('author', ''),
                'similarity': similarity
            })

    # Sort by similarity and return top results
    similarities.sort(key=lambda x: x['similarity'], reverse=True)

    # Log search analytics
    context.logger.info("Semantic search performed",
                       query=query,
                       results_count=len(similarities),
                       top_score=similarities[0]['similarity'] if similarities else 0)

    return similarities[:top_k]

async def generate_search_analytics(context):
    """Generate analytics for the search system."""

    print(f"\n=== SEARCH SYSTEM ANALYTICS ===")

    # Get all documents
    doc_nodes = await context.knowledge_graph.search_nodes({"type": "document"})

    # Get all entities
    entity_nodes = await context.knowledge_graph.search_nodes({"type": "entity"})

    # Calculate graph statistics
    stats = context.knowledge_graph.get_statistics()

    print(f"Documents indexed: {len(doc_nodes)}")
    print(f"Entities extracted: {len(entity_nodes)}")
    print(f"Total nodes: {stats['nodes']}")
    print(f"Total relationships: {stats['edges']}")

    # Category distribution
    categories = {}
    for doc in doc_nodes:
        category = doc.properties.get('category', 'unknown')
        categories[category] = categories.get(category, 0) + 1

    print(f"\nDocument categories:")
    for category, count in categories.items():
        print(f"  {category}: {count}")

    # Most frequent entities
    entity_frequencies = []
    for entity in entity_nodes:
        doc_count = entity.properties.get('document_count', 1)
        entity_frequencies.append((entity.label, doc_count))

    entity_frequencies.sort(key=lambda x: x[1], reverse=True)

    print(f"\nMost frequent entities:")
    for entity, freq in entity_frequencies[:5]:
        print(f"  {entity}: appears in {freq} documents")

    # Generate visualization if possible
    await visualize_knowledge_graph(context, doc_nodes, entity_nodes)

async def visualize_knowledge_graph(context, doc_nodes, entity_nodes):
    """Create a simple visualization of the knowledge graph."""

    try:
        # Create data for visualization
        from enterprise_ai_infrastructure.enterprise_analytics.core import BaseDataPoin
        import datetime

        # Category distribution data
        categories = {}
        for doc in doc_nodes:
            category = doc.properties.get('category', 'unknown')
            categories[category] = categories.get(category, 0) + 1

        # Create data points for visualization
        data_points = []
        timestamp = datetime.datetime.now()

        for category, count in categories.items():
            data_points.append(BaseDataPoint(
                timestamp=timestamp,
                value=count,
                source="knowledge_graph",
                source="knowledge_graph",
                metadata={
                    'metric_type': 'document_count',
                    'category': category,
                    'visualization_type': 'category_distribution'
                }
            ))

        # Generate chart using visualization engine
        if hasattr(context.analytics, 'viz_engine') and data_points:
            chart_bytes, _ = await context.analytics.viz_engine.create_chart(
                data_points,
                ChartType.PIE,
                "Document Distribution by Category",
                "Category",
                "Count"
            )

            # Save chart
            chart_path = await context.analytics.viz_engine.save_chart(
                chart_bytes,
                f"knowledge_graph_categories_{timestamp.strftime('%Y%m%d_%H%M%S')}"
            )

            print(f"\nVisualization saved: {chart_path}")

    except Exception as e:
        context.logger.debug(f"Could not generate visualization: {str(e)}")

if __name__ == "__main__":
    asyncio.run(semantic_search_example())
```

Production Deployment

Docker Deployment

dockerfile

```dockerfile
# Dockerfile
FROM python:3.11-slim

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    g++ \
    git \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Set working directory
WORKDIR /app

# Copy requirements
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY enterprise_ai_infrastructure/ ./enterprise_ai_infrastructure/
COPY config/ ./config/
COPY platform/ ./platform/
COPY examples/ ./examples/

# Create necessary directories
RUN mkdir -p /app/data /app/logs /app/cache /app/analytics_output

# Set environment variables
ENV PYTHONPATH=/app
ENV PLATFORM_ENV=production

# Expose ports
EXPOSE 8080 9090

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:8080/health || exit 1

# Run application
CMD ["python", "-m", "platform.main"]
```

yaml

```yaml
# docker-compose.yml
version: '3.8'

services:
  enterprise-ai:
    build: .
    container_name: enterprise-ai-platform
    ports:
      - "8080:8080"
      - "9090:9090"
    volumes:
      - ./data:/app/data
      - ./logs:/app/logs
      - ./config:/app/config
    environment:
      - PLATFORM_ENV=production
      - DATABASE_URL=postgresql://user:pass@postgres:5432/enterprise_ai
      - REDIS_URL=redis://redis:6379/0
    depends_on:
      - postgres
      - redis
    restart: unless-stopped

  postgres:
    image: postgres:15
    container_name: enterprise-ai-postgres
    environment:
      POSTGRES_DB: enterprise_ai
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    container_name: enterprise-ai-redis
    volumes:
      - redis_data:/data
    ports:
      - "6379:6379"
    restart: unless-stopped
  prometheus:
    image: prom/prometheus:latest
    container_name: enterprise-ai-prometheus
    ports:
      - "9091:9090"
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus
    restart: unless-stopped

  grafana:
    image: grafana/grafana:latest
    container_name: enterprise-ai-grafana
    ports:
      - "3000:3000"
    volumes:
      - grafana_data:/var/lib/grafana
      - ./monitoring/grafana:/etc/grafana/provisioning
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    restart: unless-stopped

volumes:
  postgres_data:
  redis_data:
  prometheus_data:
  grafana_data:
```

Kubernetes Deployment

yaml

```yaml
# k8s/namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: enterprise-ai
---
# k8s/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: enterprise-ai-config
  namespace: enterprise-ai
data:
  platform.yaml: |
    environment: production
    debug: false
    logger:
      min_level: INFO
      enable_console: false
      format: structured
    analytics:
      worker_pool_size: 16
      max_memory_usage_mb: 4096
    monitoring:
      enabled: true
      prometheus_enabled: true
---
# k8s/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: enterprise-ai
  namespace: enterprise-ai
spec:
  replicas: 3
  selector:
    matchLabels:
      app: enterprise-ai
  template:
    metadata:
      labels:
        app: enterprise-ai
    spec:
      containers:
```

```yaml
- name: enterprise-ai
  image: enterprise-ai:latest
  ports:
  - containerPort: 8080
  - containerPort: 9090
  env:
  - name: PLATFORM_ENV
    value: "production"
  - name: DATABASE_URL
    valueFrom:
      secretKeyRef:
        name: enterprise-ai-secrets
        key: database-url
  - name: REDIS_URL
    valueFrom:
      secretKeyRef:
        name: enterprise-ai-secrets
        key: redis-url
  volumeMounts:
  - name: config
    mountPath: /app/config
  - name: data
    mountPath: /app/data
  resources:
    requests:
      memory: "2Gi"
      cpu: "1000m"
    limits:
      memory: "4Gi"
      cpu: "2000m"
  livenessProbe:
    httpGet:
      path: /health
      port: 8080
    initialDelaySeconds: 30
    periodSeconds: 10
  readinessProbe:
    httpGet:
      path: /ready
      port: 8080
    initialDelaySeconds: 5
    periodSeconds: 5
volumes:
- name: config
  configMap:
    name: enterprise-ai-config
- name: data
```

```yaml
        persistentVolumeClaim:
          claimName: enterprise-ai-pvc
---
# k8s/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: enterprise-ai-service
  namespace: enterprise-ai
spec:
  selector:
    app: enterprise-ai
  ports:
  - name: http
    port: 80
    targetPort: 8080
  - name: metrics
    port: 9090
    targetPort: 9090
  type: ClusterIP
---
# k8s/ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: enterprise-ai-ingress
  namespace: enterprise-ai
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: enterprise-ai.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: enterprise-ai-service
            port:
              number: 80
```

Monitoring & Observability

Comprehensive Health Checks

python

```python
# platform/health.py

import asyncio
from typing import Dict, Any
from dataclasses import dataclass
from enum import Enum

class HealthStatus(Enum):
    HEALTHY = "healthy"
    WARNING = "warning"
    CRITICAL = "critical"
    UNKNOWN = "unknown"

@dataclass
class ComponentHealth:
    """Health status for a component."""
    name: str
    status: HealthStatus
    message: str
    metrics: Dict[str, Any]
    last_check: datetime.datetime

class PlatformHealthChecker:
    """Comprehensive health checking for all platform components."""

    def __init__(self, context: PlatformContext):
        self.context = context
        self.health_history = {}

    async def check_all_components(self) -> Dict[str, ComponentHealth]:
        """Check health of all platform components."""

        health_checks = await asyncio.gather(
            self._check_logger_health(),
            self._check_analytics_health(),
            self._check_knowledge_graph_health(),
            self._check_attention_engine_health(),
            self._check_system_health(),
            return_exceptions=True
        )

        component_health = {}
        check_names = ['logger', 'analytics', 'knowledge_graph', 'attention_engine', 's

        for i, result in enumerate(health_checks):
```

```python
        for i, result in enumerate(health_checks):
            if isinstance(result, Exception):
                component_health[check_names[i]] = ComponentHealth(
                    name=check_names[i],
                    status=HealthStatus.CRITICAL,
                    message=f"Health check failed: {str(result)}",
                    metrics={},
                    last_check=datetime.datetime.now()
                )
            else:
                component_health[check_names[i]] = result

        # Store in history
        self.health_history[datetime.datetime.now()] = component_health

        return component_health

    async def _check_logger_health(self) -> ComponentHealth:
        """Check logger health."""
        try:
            metrics = self.context.logger.get_metrics()

            # Determine status based on metrics
            error_rate = metrics.errorsCount / max(metrics.totalLogs, 1)

            if error_rate > 0.1:  # > 10% error rate
                status = HealthStatus.CRITICAL
                message = f"High error rate: {error_rate:.1%}"
            elif error_rate > 0.05:  # > 5% error rate
                status = HealthStatus.WARNING
                message = f"Elevated error rate: {error_rate:.1%}"
            else:
                status = HealthStatus.HEALTHY
                message = "Logger operating normally"

            return ComponentHealth(
                name="logger",
                status=status,
                message=message,
                metrics={
                    'total_logs': metrics.totalLogs,
                    'error_count': metrics.errorsCount,
                    'error_rate': error_rate,
                    'avg_processing_time_ms': metrics.avgProcessingTime,
                    'memory_usage_mb': metrics.memoryUsage / (1024 * 1024),
                    'uptime_seconds': metrics.uptime
                },
                last_check=datetime.datetime.now()
            )

        except Exception as e:
            return ComponentHealth(
                name="logger",
                status=HealthStatus.CRITICAL,
                message=f"Logger health check failed: {str(e)}",
                metrics={},
                last_check=datetime.datetime.now()
            )

    async def _check_analytics_health(self) -> ComponentHealth:
        """Check analytics engine health."""
        try:
            health_status = await self.context.analytics.health_check()

            # Convert status
            status_map = {
                'healthy': HealthStatus.HEALTHY,
                'degraded': HealthStatus.WARNING,
                'unhealthy': HealthStatus.CRITICAL
            }

            status = status_map.get(health_status.get('status', 'unknown'), HealthStatu

            return ComponentHealth(
                name="analytics",
                status=status,
                message=f"Analytics engine status: {health_status.get('status', 'unknow
                metrics=health_status.get('performance', {}),
                last_check=datetime.datetime.now()
            )

        except Exception as e:
            return ComponentHealth(
                name="analytics",
                status=HealthStatus.CRITICAL,
                message=f"Analytics health check failed: {str(e)}",
                metrics={},
                last_check=datetime.datetime.now()
            )

    async def _check_knowledge_graph_health(self) -> ComponentHealth:
        """Check knowledge graph health."""
        try:
            # Test basic operations
            test_start = time.time()

            # Test node retrieval
            test_nodes = await self.context.knowledge_graph.search_nodes({}, limit=1)

            # Test statistics
            stats = self.context.knowledge_graph.get_statistics()

            operation_time = (time.time() - test_start) * 1000  # ms

            # Determine health based on response time and stats
            if operation_time > 5000:  # > 5 seconds
                status = HealthStatus.CRITICAL
                message = f"Slow response time: {operation_time:.0f}ms"
            elif operation_time > 1000:  # > 1 second
                status = HealthStatus.WARNING
                message = f"Elevated response time: {operation_time:.0f}ms"
            else:
                status = HealthStatus.HEALTHY
                message = "Knowledge graph operating normally"

            return ComponentHealth(
                name="knowledge_graph",
                status=status,
                message=message,
                metrics={
                    'response_time_ms': operation_time,
                    'node_count': stats.get('nodes', 0),
                    'edge_count': stats.get('edges', 0),
                    'memory_usage_mb': stats.get('memory_usage_bytes', 0) / (1024 * 10
                },
                last_check=datetime.datetime.now()
            )

        except Exception as e:
            return ComponentHealth(
                name="knowledge_graph",
                status=HealthStatus.CRITICAL,
                message=f"Knowledge graph health check failed: {str(e)}",
                metrics={},
                last_check=datetime.datetime.now()
            )

    async def _check_attention_engine_health(self) -> ComponentHealth:
        """Check attention engine health."""
        try:
            # Test with small input
            import torch
            test_input = torch.randn(1, 10, 768)  # Small test tensor

            test_start = time.time()
            result = self.context.attention_pytorch.forward(test_input)
            operation_time = (time.time() - test_start) * 1000  # ms

            # Check if result is valid
            if 'output' in result and result['output'] is not None:
                status = HealthStatus.HEALTHY
                message = "Attention engine operating normally"
            else:
                status = HealthStatus.WARNING
                message = "Attention engine returned unexpected output"

            return ComponentHealth(
                name="attention_engine",
                status=status,
                message=message,
                metrics={
                    'response_time_ms': operation_time,
                    'output_shape': list(result['output'].shape) if 'output' in result
                    'framework': 'pytorch'
                },
                last_check=datetime.datetime.now()
            )

        except Exception as e:
            return ComponentHealth(
                name="attention_engine",
                status=HealthStatus.CRITICAL,
                message=f"Attention engine health check failed: {str(e)}",
                metrics={},
                last_check=datetime.datetime.now()
            )

    async def _check_system_health(self) -> ComponentHealth:
        """Check system resource health."""
        try:
            import psutil

            # Get system metrics
            cpu_percent = psutil.cpu_percent(interval=1)
            memory = psutil.virtual_memory()
            disk = psutil.disk_usage('/')
            # Determine overall system health
            critical_conditions = []
            warning_conditions = []

            if cpu_percent > 90:
                critical_conditions.append(f"CPU: {cpu_percent:.1f}%")
            elif cpu_percent > 75:
                warning_conditions.append(f"CPU: {cpu_percent:.1f}%")

            if memory.percent > 95:
                critical_conditions.append(f"Memory: {memory.percent:.1f}%")
            elif memory.percent > 85:
                warning_conditions.append(f"Memory: {memory.percent:.1f}%")

            if disk.percent > 95:
                critical_conditions.append(f"Disk: {disk.percent:.1f}%")
            elif disk.percent > 85:
                warning_conditions.append(f"Disk: {disk.percent:.1f}%")

            # Determine status
            if critical_conditions:
                status = HealthStatus.CRITICAL
                message = f"Critical resource usage: {', '.join(critical_conditions)}"
            elif warning_conditions:
                status = HealthStatus.WARNING
                message = f"High resource usage: {', '.join(warning_conditions)}"
            else:
                status = HealthStatus.HEALTHY
                message = "System resources normal"

            return ComponentHealth(
                name="system",
                status=status,
                message=message,
                metrics={
                    'cpu_percent': cpu_percent,
                    'memory_percent': memory.percent,
                    'memory_available_gb': memory.available / (1024**3),
                    'disk_percent': disk.percent,
                    'disk_free_gb': disk.free / (1024**3)
                },
                last_check=datetime.datetime.now()
            )

        except Exception as e:
            return ComponentHealth(
                name="system",
                status=HealthStatus.CRITICAL,
                message=f"System health check failed: {str(e)}",
                metrics={},
                last_check=datetime.datetime.now()
            )

    def get_overall_health(self, component_health: Dict[str, ComponentHealth]) -> Healt
        """Determine overall platform health."""
        statuses = [health.status for health in component_health.values()]

        if HealthStatus.CRITICAL in statuses:
            return HealthStatus.CRITICAL
        elif HealthStatus.WARNING in statuses:
            return HealthStatus.WARNING
        elif all(status == HealthStatus.HEALTHY for status in statuses):
            return HealthStatus.HEALTHY
        else:
            return HealthStatus.UNKNOWN
```

Metrics Collection & Prometheus Integration

python

```python
# platform/metrics.py

import time
from typing import Dict, Any, Optional
from prometheus_client import Counter, Histogram, Gauge, CollectorRegistry, generate_la

class PlatformMetrics:
    """Prometheus metrics for the platform."""

    def __init__(self):
        self.registry = CollectorRegistry()

        # Request metrics
        self.request_count = Counter(
            'platform_requests_total',
            'Total number of requests',
            ['method', 'endpoint', 'status'],
            registry=self.registry
        )

        self.request_duration = Histogram(
            'platform_request_duration_seconds',
            'Request duration in seconds',
            ['method', 'endpoint'],
            registry=self.registry
        )

        # Component metrics
        self.component_health = Gauge(
            'platform_component_health',
            'Component health status (1=healthy, 0.5=warning, 0=critical)',
            ['component'],
            registry=self.registry
        )

        self.analytics_data_points = Counter(
            'platform_analytics_data_points_total',
            'Total analytics data points processed',
            ['source'],
            registry=self.registry
        )

        self.knowledge_graph_operations = Counter(
            'platform_kg_operations_total',
            'Total knowledge graph operations',
            ['operation_type'],
            registry=self.registry
        )

        self.attention_computations = Counter(
            'platform_attention_computations_total',
            'Total attention computations',
            ['attention_type'],
            registry=self.registry
        )

        # Resource metrics
        self.memory_usage = Gauge(
            'platform_memory_usage_bytes',
            'Memory usage in bytes',
            ['component'],
            registry=self.registry
        )

        self.cpu_usage = Gauge(
            'platform_cpu_usage_percent',
            'CPU usage percentage',
            ['component'],
            registry=self.registry
        )

    def record_request(self, method: str, endpoint: str, status: int, duration: float):
        """Record request metrics."""
        self.request_count.labels(method=method, endpoint=endpoint, status=str(status)
        self.request_duration.labels(method=method, endpoint=endpoint).observe(duratio

    def update_component_health(self, component: str, health_status: str):
        """Update component health metrics."""
        health_value = {
            'healthy': 1.0,
            'warning': 0.5,
            'critical': 0.0,
            'unknown': -1.0
        }.get(health_status, -1.0)

        self.component_health.labels(component=component).set(health_value)

    def record_analytics_data_point(self, source: str):
        """Record analytics data point."""
        self.analytics_data_points.labels(source=source).inc()
    def record_kg_operation(self, operation_type: str):
        """Record knowledge graph operation."""
        self.knowledge_graph_operations.labels(operation_type=operation_type).inc()

    def record_attention_computation(self, attention_type: str):
        """Record attention computation."""
        self.attention_computations.labels(attention_type=attention_type).inc()

    def update_resource_usage(self, component: str, memory_bytes: int, cpu_percent: fl
        """Update resource usage metrics."""
        self.memory_usage.labels(component=component).set(memory_bytes)
        self.cpu_usage.labels(component=component).set(cpu_percent)

    def get_metrics(self) -> str:
        """Get metrics in Prometheus format."""
        return generate_latest(self.registry).decode('utf-8')

# Global metrics instance
platform_metrics = PlatformMetrics()
```

Troubleshooting

Common Issues and Solutions

1. Memory Issues

Symptoms:

- High memory usage
- Out of memory errors
- Slow performance

Solutions:

```python
# Memory optimization configuration
config = PlatformConfig()

# Reduce batch sizes
config.analytics["batch_size"] = 500
config.logger["batch_size"] = 250

# Enable memory cleanup
config.analytics["enable_streaming"] = True
config.knowledge_graph["memory_limit_gb"] = 2

# Use smaller attention configurations
config.attention["hidden_size"] = 512  # Instead of 768
config.attention["max_sequence_length"] = 1024  # Instead of 2048
```

2. Performance Issues

Symptoms:

- Slow response times
- High CPU usage
- Timeouts

Solutions:

```python
# Performance optimization
config.analytics["worker_pool_size"] = 16  # Increase workers
config.knowledge_graph["max_workers"] = 12
config.attention["use_flash_attention"] = True  # Enable Flash Attention
config.attention["memory_efficient"] = True
```

3. Integration Issues

Symptoms:

- Module communication failures
- Inconsistent data
- Missing correlation

Solutions:

python

```python
# Enable comprehensive logging and correlation
config.logger["enable_correlation"] = True
config.logger["min_level"] = "DEBUG"

# Use platform context for all operations
async with context.correlation_context() as correlation_id:
    # All operations will be correlated
    await process_data(data)
```

Debugging Workflow

python

```python
# platform/debug.py

class PlatformDebugger:
    """Debugging utilities for the platform."""

    def __init__(self, context: PlatformContext):
        self.context = context

    async def run_diagnostic(self) -> Dict[str, Any]:
        """Run comprehensive platform diagnostic."""

        diagnostic = {
            'timestamp': datetime.datetime.now().isoformat(),
            'configuration': self._get_config_summary(),
            'health_checks': await self._run_health_checks(),
            'performance_metrics': await self._collect_performance_metrics(),
            'integration_tests': await self._run_integration_tests(),
            'recommendations': []
        }

        # Generate recommendations
        diagnostic['recommendations'] = self._generate_recommendations(diagnostic)

        return diagnostic

    def _get_config_summary(self) -> Dict[str, Any]:
        """Get configuration summary."""
        return {
            'environment': self.context.config.environment,
            'debug_mode': self.context.config.debug,
            'worker_pools': {
                'analytics': self.context.config.analytics['worker_pool_size'],
                'knowledge_graph': self.context.config.knowledge_graph['max_workers']
            },
            'memory_limits': {
                'analytics': self.context.config.analytics['max_memory_usage_mb'],
                'knowledge_graph': self.context.config.knowledge_graph['memory_limit_gb
            }
        }

    async def _run_health_checks(self) -> Dict[str, Any]:
        """Run all health checks."""
        health_checker = PlatformHealthChecker(self.context)
        component_health = await health_checker.check_all_components()
        return {
            'overall_status': health_checker.get_overall_health(component_health).valu
            'components': {
                name: {
                    'status': health.status.value,
                    'message': health.message,
                    'metrics': health.metrics
                }
                for name, health in component_health.items()
            }
        }

    async def _collect_performance_metrics(self) -> Dict[str, Any]:
        """Collect performance metrics from all components."""

        # Analytics performance
        analytics_metrics = await self.context.analytics.get_performance_metrics()

        # Logger performance
        logger_metrics = self.context.logger.get_metrics()

        # Knowledge graph stats
        kg_stats = self.context.knowledge_graph.get_statistics()

        return {
            'analytics': analytics_metrics,
            'logger': {
                'total_logs': logger_metrics.totalLogs,
                'avg_processing_time_ms': logger_metrics.avgProcessingTime,
                'error_rate': logger_metrics.errorsCount / max(logger_metrics.totalLog
            },
            'knowledge_graph': kg_stats,
            'system': self._get_system_metrics()
        }

    def _get_system_metrics(self) -> Dict[str, Any]:
        """Get system performance metrics."""
        try:
            import psutil
            return {
                'cpu_percent': psutil.cpu_percent(),
                'memory_percent': psutil.virtual_memory().percent,
                'disk_percent': psutil.disk_usage('/').percent,
                'load_average': psutil.getloadavg() if hasattr(psutil, 'getloadavg') e
            }
        except:
            return {'error': 'Could not collect system metrics'}

    async def _run_integration_tests(self) -> Dict[str, Any]:
        """Run integration tests between components."""

        tests = {}

        # Test 1: Logger → Analytics integration
        try:
            # Generate test log
            test_correlation_id = f"test-{int(time.time())}"
            self.context.log("info", "Integration test message",
                           test_data={"value": 42},
                           correlation_id=test_correlation_id)

            # Wait for processing
            await asyncio.sleep(1)

            tests['logger_analytics'] = {
                'status': 'passed',
                'message': 'Logger to analytics integration working'
            }
        except Exception as e:
            tests['logger_analytics'] = {
                'status': 'failed',
                'message': f'Logger to analytics integration failed: {str(e)}'
            }

        # Test 2: Knowledge Graph → Analytics integration
        try:
            # Create test node
            from enterprise_ai_infrastructure.enterprise_knowledge_graph import GraphNc
            test_node = GraphNode(
                id=f"test_node_{int(time.time())}",
                type=NodeType.ENTITY,
                label="Integration Test Node",
                properties={"test": True, "created_by": "integration_test"}
            )

            await self.context.knowledge_graph.add_node(test_node)

            # Verify node exists
            retrieved_node = await self.context.knowledge_graph.get_node(test_node.id)

            if retrieved_node:
                tests['knowledge_graph_storage'] = {
```

```python
                    'status': 'passed',
                    'message': 'Knowledge graph storage working'
                }

                # Clean up test node
                await self.context.knowledge_graph.remove_node(test_node.id)
            else:
                tests['knowledge_graph_storage'] = {
                    'status': 'failed',
                    'message': 'Could not retrieve stored node'
                }

        except Exception as e:
            tests['knowledge_graph_storage'] = {
                'status': 'failed',
                'message': f'Knowledge graph storage failed: {str(e)}'
            }

        # Test 3: Attention Engine functionality
        try:
            import torch
            test_input = torch.randn(1, 10, 768)
            result = self.context.attention_pytorch.forward(test_input)

            if 'output' in result and result['output'] is not None:
                tests['attention_engine'] = {
                    'status': 'passed',
                    'message': 'Attention engine processing working',
                    'output_shape': list(result['output'].shape)
                }
            else:
                tests['attention_engine'] = {
                    'status': 'failed',
                    'message': 'Attention engine returned invalid output'
                }

        except Exception as e:
            tests['attention_engine'] = {
                'status': 'failed',
                'message': f'Attention engine test failed: {str(e)}'
            }

        return tests

    def _generate_recommendations(self, diagnostic: Dict[str, Any]) -> List[str]:
        """Generate recommendations based on diagnostic results."""
        recommendations = []

        # Check overall health
        if diagnostic['health_checks']['overall_status'] != 'healthy':
            recommendations.append("Address component health issues before proceeding"

        # Check performance
        perf = diagnostic['performance_metrics']

        # System resources
        if 'system' in perf and 'cpu_percent' in perf['system']:
            cpu = perf['system']['cpu_percent']
            memory = perf['system']['memory_percent']

            if cpu > 80:
                recommendations.append(f"High CPU usage ({cpu:.1f}%) - consider scalin
            if memory > 85:
                recommendations.append(f"High memory usage ({memory:.1f}%) - review me

        # Logger performance
        if 'logger' in perf:
            error_rate = perf['logger']['error_rate']
            if error_rate > 0.05:
                recommendations.append(f"High logger error rate ({error_rate:.1%}) - c

        # Knowledge graph size
        if 'knowledge_graph' in perf:
            nodes = perf['knowledge_graph'].get('nodes', 0)
            if nodes > 100000:
                recommendations.append("Large knowledge graph - consider partitioning

        # Integration test failures
        failed_tests = [
            name for name, result in diagnostic['integration_tests'].items()
            if result['status'] == 'failed'
        ]

        if failed_tests:
            recommendations.append(f"Failed integration tests: {', '.join(failed_tests

        # Configuration recommendations
        config = diagnostic['configuration']
        if config['worker_pools']['analytics'] < 4:
            recommendations.append("Consider increasing analytics worker pool size for

        if not recommendations:
```

```python
            recommendations.append("Platform is operating optimally")

        return recommendations

# Diagnostic CLI command
async def run_platform_diagnostic():
    """CLI command to run platform diagnostic."""

    from config.platform_config import get_config
    from platform.context import PlatformContext

    config = get_config()
    context = PlatformContext(config)

    try:
        await context.initialize()

        debugger = PlatformDebugger(context)
        diagnostic = await debugger.run_diagnostic()

        print("\n=== PLATFORM DIAGNOSTIC REPORT ===")
        print(f"Timestamp: {diagnostic['timestamp']}")
        print(f"Environment: {diagnostic['configuration']['environment']}")

        print(f"\n--- OVERALL HEALTH ---")
        print(f"Status: {diagnostic['health_checks']['overall_status'].upper()}")

        print(f"\n--- COMPONENT HEALTH ---")
        for component, health in diagnostic['health_checks']['components'].items():
            status_indicator = {
                'healthy': 'OK',
                'warning': 'WARN',
                'critical': 'FAIL',
                'unknown': 'UNKNOWN'
            }.get(health['status'], 'UNKNOWN')

            print(f"[{status_indicator}] {component}: {health['message']}")

        print(f"\n--- INTEGRATION TESTS ---")
        for test_name, result in diagnostic['integration_tests'].items():
            status_indicator = 'PASS' if result['status'] == 'passed' else 'FAIL'
            print(f"[{status_indicator}] {test_name}: {result['message']}")

        print(f"\n--- RECOMMENDATIONS ---")
        for rec in diagnostic['recommendations']:
            print(f"  - {rec}")
        print(f"\n--- PERFORMANCE SUMMARY ---")
        perf = diagnostic['performance_metrics']
        if 'system' in perf:
            sys_metrics = perf['system']
            print(f"  CPU: {sys_metrics.get('cpu_percent', 'N/A'):.1f}%")
            print(f"  Memory: {sys_metrics.get('memory_percent', 'N/A'):.1f}%")
            print(f"  Disk: {sys_metrics.get('disk_percent', 'N/A'):.1f}%")

        if 'analytics' in perf:
            print(f"  Analytics uptime: {perf['analytics'].get('uptime_seconds', 0):.0f

        if 'knowledge_graph' in perf:
            kg_stats = perf['knowledge_graph']
            print(f"  Knowledge Graph: {kg_stats.get('nodes', 0)} nodes, {kg_stats.get(

    finally:
        await context.close()

if __name__ == "__main__":
    asyncio.run(run_platform_diagnostic())
```

Performance Optimization

Memory Optimization

python

```python
# platform/optimization.py

class MemoryOptimizer:
    """Memory optimization utilities for the platform."""

    def __init__(self, context: PlatformContext):
        self.context = context

    async def optimize_memory_usage(self) -> Dict[str, Any]:
        """Optimize memory usage across all components."""

        optimizations = {}

        # 1. Logger optimization
        logger_opts = await self._optimize_logger_memory()
        optimizations['logger'] = logger_opts

        # 2. Analytics optimization
        analytics_opts = await self._optimize_analytics_memory()
        optimizations['analytics'] = analytics_opts

        # 3. Knowledge graph optimization
        kg_opts = await self._optimize_knowledge_graph_memory()
        optimizations['knowledge_graph'] = kg_opts

        # 4. Attention engine optimization
        attention_opts = await self._optimize_attention_memory()
        optimizations['attention_engine'] = attention_opts

        # 5. System-wide optimization
        system_opts = await self._optimize_system_memory()
        optimizations['system'] = system_opts

        return optimizations

    async def _optimize_logger_memory(self) -> Dict[str, Any]:
        """Optimize logger memory usage."""

        # Force flush of logger buffers
        await self.context.logger._flush_buffer()

        # Get current metrics
        metrics = self.context.logger.get_metrics()
        initial_memory = metrics.memoryUsage
        # Optimization recommendations
        recommendations = []

        if metrics.totalLogs > 10000:
            recommendations.append("Consider increasing flush interval to reduce memory

        if metrics.avgProcessingTime > 100:  # > 100ms
            recommendations.append("Enable worker threads for CPU-intensive logging ope

        return {
            'initial_memory_mb': initial_memory / (1024 * 1024),
            'recommendations': recommendations,
            'actions_taken': ['force_buffer_flush']
        }

    async def _optimize_analytics_memory(self) -> Dict[str, Any]:
        """Optimize analytics memory usage."""

        actions_taken = []

        # Clear analytics cache if it exists
        if hasattr(self.context.analytics, '_cache_manager'):
            await self.context.analytics._cache_manager.invalidate()
            actions_taken.append('cleared_cache')

        # Get performance metrics
        perf_metrics = await self.context.analytics.get_performance_metrics()

        recommendations = []
        if perf_metrics.get('memory_usage_mb', 0) > 1000:  # > 1GB
            recommendations.append("Consider reducing batch size or enabling streaming

        return {
            'actions_taken': actions_taken,
            'recommendations': recommendations,
            'performance_metrics': perf_metrics
        }

    async def _optimize_knowledge_graph_memory(self) -> Dict[str, Any]:
        """Optimize knowledge graph memory usage."""

        stats = self.context.knowledge_graph.get_statistics()
        initial_memory = stats.get('memory_usage_bytes', 0)

        actions_taken = []
        recommendations = []
        # Check if graph is large
        if stats.get('nodes', 0) > 50000:
            recommendations.append("Consider implementing graph partitioning for large

        if stats.get('edges', 0) > 100000:
            recommendations.append("Enable edge compression for large graphs")

        # Force garbage collection on the knowledge graph
        import gc
        gc.collect()
        actions_taken.append('forced_garbage_collection')

        return {
            'initial_memory_mb': initial_memory / (1024 * 1024),
            'node_count': stats.get('nodes', 0),
            'edge_count': stats.get('edges', 0),
            'actions_taken': actions_taken,
            'recommendations': recommendations
        }

    async def _optimize_attention_memory(self) -> Dict[str, Any]:
        """Optimize attention engine memory usage."""

        recommendations = []

        # Check current configuration
        config = self.context.config.attention

        if config['hidden_size'] > 1024:
            recommendations.append("Consider reducing hidden_size for memory-constrain

        if config['max_sequence_length'] > 2048:
            recommendations.append("Enable Flash Attention for long sequences to reduc

        if not config.get('use_flash_attention', False):
            recommendations.append("Enable Flash Attention to reduce memory complexity

        return {
            'current_config': {
                'hidden_size': config['hidden_size'],
                'max_sequence_length': config['max_sequence_length'],
                'flash_attention_enabled': config.get('use_flash_attention', False)
            },
            'recommendations': recommendations
        }
```

```python
    async def _optimize_system_memory(self) -> Dict[str, Any]:
        """Optimize system-wide memory usage."""

        import psutil
        import gc

        # Get initial memory
        initial_memory = psutil.virtual_memory()

        # Force garbage collection
        collected = gc.collect()

        # Get final memory
        final_memory = psutil.virtual_memory()

        memory_freed = initial_memory.used - final_memory.used

        recommendations = []

        if final_memory.percent > 85:
            recommendations.append("System memory usage is high - consider scaling hor:

        if final_memory.available < 1024 * 1024 * 1024:  # < 1GB available
            recommendations.append("Low available memory - immediate optimization requ:

        return {
            'initial_memory_percent': initial_memory.percent,
            'final_memory_percent': final_memory.percent,
            'memory_freed_mb': memory_freed / (1024 * 1024),
            'garbage_objects_collected': collected,
            'recommendations': recommendations,
            'actions_taken': ['forced_garbage_collection']
        }

class PerformanceOptimizer:
    """Performance optimization utilities."""

    def __init__(self, context: PlatformContext):
        self.context = context
        self.benchmarks = {}

    async def run_performance_benchmarks(self) -> Dict[str, Any]:
        """Run comprehensive performance benchmarks."""

        benchmarks = {}
        # 1. Logger benchmark
        benchmarks['logger'] = await self._benchmark_logger()

        # 2. Analytics benchmark
        benchmarks['analytics'] = await self._benchmark_analytics()

        # 3. Knowledge graph benchmark
        benchmarks['knowledge_graph'] = await self._benchmark_knowledge_graph()

        # 4. Attention engine benchmark
        benchmarks['attention_engine'] = await self._benchmark_attention_engine()

        # 5. End-to-end pipeline benchmark
        benchmarks['end_to_end'] = await self._benchmark_end_to_end()

        return benchmarks

    async def _benchmark_logger(self) -> Dict[str, Any]:
        """Benchmark logger performance."""

        # Test batch logging performance
        import time

        num_logs = 1000
        start_time = time.perf_counter()

        for i in range(num_logs):
            self.context.logger.info(f"Benchmark log message {i}",
                                     iteration=i,
                                     benchmark=True)

        # Force flush to measure total time
        await self.context.logger._flush_buffer()

        end_time = time.perf_counter()
        total_time = end_time - start_time

        return {
            'total_logs': num_logs,
            'total_time_seconds': total_time,
            'logs_per_second': num_logs / total_time,
            'avg_time_per_log_ms': (total_time / num_logs) * 1000
        }

    async def _benchmark_analytics(self) -> Dict[str, Any]:
        """Benchmark analytics performance."""
        from enterprise_ai_infrastructure.enterprise_analytics.core import BaseDataPoir
        import datetime
        import time

        # Create test data points
        num_points = 1000
        timestamp = datetime.datetime.now()

        data_points = []
        for i in range(num_points):
            data_points.append(BaseDataPoint(
                timestamp=timestamp + datetime.timedelta(seconds=i),
                value=random.uniform(0, 100),
                source="benchmark",
                metadata={'iteration': i, 'benchmark': True}
            ))

        # Benchmark data processing
        start_time = time.perf_counter()

        # Process through any available processors
        processed_data = data_points
        for processor in self.context.analytics._processors:
            processed_data = await processor.process(processed_data)

        end_time = time.perf_counter()
        total_time = end_time - start_time

        return {
            'data_points_processed': len(processed_data),
            'total_time_seconds': total_time,
            'points_per_second': len(processed_data) / total_time,
            'avg_time_per_point_ms': (total_time / len(processed_data)) * 1000
        }

    async def _benchmark_knowledge_graph(self) -> Dict[str, Any]:
        """Benchmark knowledge graph performance."""

        import time
        from enterprise_ai_infrastructure.enterprise_knowledge_graph import GraphNode,

        # Benchmark node creation
        num_nodes = 100

        start_time = time.perf_counter()
```

```python
        created_nodes = []
        for i in range(num_nodes):
            node = GraphNode(
                id=f"benchmark_node_{i}",
                type=NodeType.ENTITY,
                label=f"Benchmark Node {i}",
                properties={'benchmark': True, 'iteration': i}
            )

            await self.context.knowledge_graph.add_node(node)
            created_nodes.append(node.id)

        creation_time = time.perf_counter() - start_time

        # Benchmark node retrieval
        start_time = time.perf_counter()

        for node_id in created_nodes:
            await self.context.knowledge_graph.get_node(node_id)

        retrieval_time = time.perf_counter() - start_time

        # Benchmark search
        start_time = time.perf_counter()
        search_results = await self.context.knowledge_graph.search_nodes({'benchmark':
        search_time = time.perf_counter() - start_time

        # Clean up benchmark nodes
        for node_id in created_nodes:
            await self.context.knowledge_graph.remove_node(node_id)

        return {
            'nodes_created': num_nodes,
            'creation_time_seconds': creation_time,
            'creation_rate_nodes_per_second': num_nodes / creation_time,
            'retrieval_time_seconds': retrieval_time,
            'retrieval_rate_nodes_per_second': num_nodes / retrieval_time,
            'search_time_seconds': search_time,
            'search_results_found': len(search_results)
        }

    async def _benchmark_attention_engine(self) -> Dict[str, Any]:
        """Benchmark attention engine performance."""

        import torch
        import time
        # Test different input sizes
        test_sizes = [
            (1, 32, 768),   # Small
            (1, 128, 768),  # Medium
            (1, 512, 768),  # Large
        ]

        results = {}

        for batch, seq_len, hidden_size in test_sizes:
            size_key = f"{batch}x{seq_len}x{hidden_size}"

            # Create test input
            test_input = torch.randn(batch, seq_len, hidden_size)

            # Warm up
            for _ in range(3):
                _ = self.context.attention_pytorch.forward(test_input)

            # Benchmark
            num_runs = 10
            start_time = time.perf_counter()

            for _ in range(num_runs):
                result = self.context.attention_pytorch.forward(test_input)

            end_time = time.perf_counter()
            total_time = end_time - start_time

            results[size_key] = {
                'input_shape': list(test_input.shape),
                'total_time_seconds': total_time,
                'avg_time_per_forward_ms': (total_time / num_runs) * 1000,
                'throughput_forwards_per_second': num_runs / total_time,
                'tokens_per_second': (batch * seq_len * num_runs) / total_time
            }

        return results

    async def _benchmark_end_to_end(self) -> Dict[str, Any]:
        """Benchmark end-to-end pipeline performance."""

        from platform.pipeline import IntegratedPipeline, TextProcessingStage, Knowledg
        import time
```

```
# Create pipeline
pipeline = IntegratedPipeline(self.context)
pipeline.add_stage(TextProcessingStage("text_processing", self.context))
pipeline.add_stage(KnowledgeExtractionStage("knowledge_extraction", self.conte
pipeline.add_stage(AnalyticsStage("analytics", self.context))

# Test documents
test_documents = [
    "Apple Inc. is a technology company based in Cupertino, California.",
    "Google develops artificial intelligence and machine learning technologies.
    "Microsoft Azure provides cloud computing services for enterprises.",
    "Amazon Web Services offers scalable infrastructure solutions.",
    "Tesla manufactures electric vehicles and sustainable energy products."
]

# Benchmark pipeline processing
start_time = time.perf_counter()

results = []
for doc in test_documents:
    result = await pipeline.process(doc)
    results.append(result)

end_time = time.perf_counter()
total_time = end_time - start_time

# Calculate statistics
total_entities = sum(
    len(result['result'].get('entities_extracted', []))
    for result in results
)

total_nodes = sum(
    len(result['result'].get('nodes_created', []))
    for result in results
)

return {
    'documents_processed': len(test_documents),
    'total_time_seconds': total_time,
    'avg_time_per_document_seconds': total_time / len(test_documents),
    'documents_per_second': len(test_documents) / total_time,
    'total_entities_extracted': total_entities,
    'total_nodes_created': total_nodes,
    'entities_per_second': total_entities / total_time,
    'nodes_per_second': total_nodes / total_time
}
```

```python
# Performance optimization CLI command
async def optimize_platform_performance():
    """CLI command to optimize platform performance."""

    from config.platform_config import get_config
    from platform.context import PlatformContext

    config = get_config()
    context = PlatformContext(config)

    try:
        await context.initialize()

        print("\n=== PLATFORM PERFORMANCE OPTIMIZATION ===")

        # Memory optimization
        print("\n--- MEMORY OPTIMIZATION ---")
        memory_optimizer = MemoryOptimizer(context)
        memory_results = await memory_optimizer.optimize_memory_usage()

        for component, result in memory_results.items():
            print(f"\n{component.upper()}:")
            if 'actions_taken' in result:
                for action in result['actions_taken']:
                    print(f"  [ACTION] {action}")
            if 'recommendations' in result:
                for rec in result['recommendations']:
                    print(f"  [RECOMMENDATION] {rec}")

        # Performance benchmarks
        print("\n--- PERFORMANCE BENCHMARKS ---")
        performance_optimizer = PerformanceOptimizer(context)
        benchmark_results = await performance_optimizer.run_performance_benchmarks()

        for component, results in benchmark_results.items():
            print(f"\n{component.upper()}:")
            if isinstance(results, dict):
                for metric, value in results.items():
                    if isinstance(value, float):
                        print(f"  {metric}: {value:.3f}")
                    else:
                        print(f"  {metric}: {value}")

        # Generate optimization recommendations
        print("\n--- OPTIMIZATION RECOMMENDATIONS ---")
        # Check benchmark results for optimization opportunities
        logger_perf = benchmark_results.get('logger', {})
        if logger_perf.get('logs_per_second', 0) < 1000:
            print("  - Consider enabling batching to improve logger throughput")

        kg_perf = benchmark_results.get('knowledge_graph', {})
        if kg_perf.get('creation_rate_nodes_per_second', 0) < 100:
            print("  - Consider optimizing knowledge graph indexing strategy")

        attention_perf = benchmark_results.get('attention_engine', {})
        for size, metrics in attention_perf.items():
            if metrics.get('avg_time_per_forward_ms', 0) > 100:
                print(f"  - Attention engine slow for {size} - consider Flash Attentio

        print("\n=== OPTIMIZATION COMPLETE ===")

    finally:
        await context.close()

if __name__ == "__main__":
    asyncio.run(optimize_platform_performance())
```

Best Practices

1. Configuration Management

python

```python
# Best practices for configuration

# DO: Use environment-specific configurations
config = PlatformConfig.from_file(Path(f"config/environments/{os.getenv('PLATFORM_ENV'

# DO: Validate configuration on startup
try:
    context = PlatformContext(config)
    await context.initialize()
except Exception as e:
    logger.fatal("Configuration validation failed", error=e)
    exit(1)

# DO: Use correlation IDs for tracking
async with context.correlation_context() as correlation_id:
    result = await process_request(data)

# AVOID: Hardcoded configurations
# config.analytics["worker_pool_size"] = 8  # Bad

# DO: Environment-aware configurations
config.analytics["worker_pool_size"] = int(os.getenv("ANALYTICS_WORKERS", "8"))
```

2. Error Handling

python

```python
# Best practices for error handling

# DO: Use structured error logging
try:
    result = await risky_operation()
except SpecificException as e:
    context.log("error", "Operation failed with known error",
               error_type=type(e).__name__,
               operation="risky_operation",
               retry_count=retry_count)
    # Handle specific error
except Exception as e:
    context.log("error", "Unexpected error in operation",
               error_type=type(e).__name__,
               operation="risky_operation")
    raise  # Re-raise unexpected errors

# DO: Implement circuit breakers for external dependencies
class CircuitBreaker:
    def __init__(self, failure_threshold=5, recovery_timeout=60):
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.failure_count = 0
        self.last_failure = None
        self.state = "closed"  # closed, open, half-open

# DO: Use retries with exponential backoff
@retry_on_failure(max_attempts=3, backoff_factor=2.0)
async def external_api_call():
    # Implementation
    pass
```

3. Performance Optimization

python

```python
# Best practices for performance

# DO: Use batching for high-throughput operations
async def process_data_batch(data_items: List[Any], batch_size: int = 1000):
    for i in range(0, len(data_items), batch_size):
        batch = data_items[i:i + batch_size]
        await process_batch(batch)

# DO: Implement proper caching strategies
@lru_cache(maxsize=1000)
def expensive_computation(input_data: str) -> str:
    # Cached computation
    return result

# DO: Use streaming for large datasets
async def process_large_dataset(data_source):
    async for batch in data_source.stream_batches(batch_size=1000):
        await process_batch(batch)
        # Allow other tasks to run
        await asyncio.sleep(0)

# DO: Monitor resource usage
async def memory_aware_processing(data):
    import psutil
    if psutil.virtual_memory().percent > 85:
        await asyncio.sleep(1)  # Back pressure

    return await process_data(data)
```

4. Security Practices

python

```python
# Best practices for security

# DO: Sanitize all inputs
def sanitize_input(user_input: str) -> str:
    # Remove potentially dangerous characters
    safe_input = re.sub(r'[<>\"\'&]', '', user_input)
    return safe_input[:1000]  # Limit length

# DO: Use environment variables for secrets
DATABASE_URL = os.getenv("DATABASE_URL")
if not DATABASE_URL:
    raise ValueError("DATABASE_URL environment variable required")

# DO: Implement rate limiting
from collections import defaultdict
import time

class RateLimiter:
    def __init__(self, max_requests: int = 100, window_seconds: int = 60):
        self.max_requests = max_requests
        self.window_seconds = window_seconds
        self.requests = defaultdict(list)

    def is_allowed(self, client_id: str) -> bool:
        now = time.time()
        client_requests = self.requests[client_id]

        # Remove old requests
        client_requests[:] = [req_time for req_time in client_requests
                             if now - req_time < self.window_seconds]

        if len(client_requests) >= self.max_requests:
            return False

        client_requests.append(now)
        return True

# DO: Log security events
context.log("warn", "Suspicious activity detected",
           client_ip=request.client_ip,
           endpoint=request.endpoint,
           user_agent=request.headers.get("user-agent"))
```

5. Testing Strategies

```python
# Best practices for testing

# DO: Use dependency injection for testability
class TestableService:
    def __init__(self, context: PlatformContext, external_api: ExternalAPI):
        self.context = context
        self.external_api = external_api

# DO: Create integration tests
async def test_full_pipeline():
    config = PlatformConfig()
    config.environment = "test"

    context = PlatformContext(config)
    await context.initialize()

    try:
        # Test full pipeline
        result = await process_document("test document")
        assert result['status'] == 'success'
        assert len(result['entities']) > 0
    finally:
        await context.close()

# DO: Mock external dependencies
class MockExternalAPI:
    async def get_data(self):
        return {"test": "data"}

# DO: Test error conditions
async def test_error_handling():
    with pytest.raises(SpecificException):
        await operation_that_should_fail()
```

6. Monitoring and Observability

python

```python
# Best practices for monitoring

# DO: Use structured logging
context.log("info", "Processing request",
           request_id=request_id,
           user_id=user_id,
           operation="document_processing",
           duration_ms=processing_time)

# DO: Implement health checks
async def health_check():
    checks = {
        'database': await check_database_connection(),
        'redis': await check_redis_connection(),
        'external_apis': await check_external_apis()
    }

    overall_health = all(checks.values())
    return {
        'status': 'healthy' if overall_health else 'unhealthy',
        'checks': checks,
        'timestamp': datetime.datetime.now().isoformat()
    }

# DO: Track business metrics
context.log("info", "Document processed successfully",
           document_type=doc_type,
           processing_time_ms=duration,
           entities_extracted=entity_count,
           user_id=user_id)

# DO: Set up alerts for critical metrics
if error_rate > 0.05:  # > 5% error rate
    send_alert("High error rate detected",
              severity="high",
              metric="error_rate",
              value=error_rate)
```

7. Deployment Best Practices

python

```python
# Best practices for deployment

# DO: Use graceful shutdown
import signal

class GracefulShutdown:
    def __init__(self, context: PlatformContext):
        self.context = context
        self.shutdown_event = asyncio.Event()

        # Register signal handlers
        signal.signal(signal.SIGTERM, self._signal_handler)
        signal.signal(signal.SIGINT, self._signal_handler)

    def _signal_handler(self, signum, frame):
        self.context.log("info", f"Received signal {signum}, starting graceful shutdowr
        self.shutdown_event.set()

    async def wait_for_shutdown(self):
        await self.shutdown_event.wait()

        # Graceful shutdown sequence
        self.context.log("info", "Starting graceful shutdown")

        # Stop accepting new requests
        # Complete in-flight requests
        # Close connections
        await self.context.close()

        self.context.log("info", "Graceful shutdown completed")

# DO: Implement readiness and liveness probes
async def readiness_check():
    """Check if service is ready to accept traffic."""
    try:
        # Check all dependencies
        await context.analytics.health_check()
        await context.knowledge_graph.get_statistics()
        return {"status": "ready"}
    except:
        return {"status": "not_ready"}

async def liveness_check():
    """Check if service is alive."""
    try:
```

```python
    try:
        # Basic health check
        return {"status": "alive", "timestamp": datetime.datetime.now().isoformat()}
    except:
        return {"status": "not_alive"}

# DO: Use feature flags
class FeatureFlags:
    def __init__(self):
        self.flags = {
            'enable_new_algorithm': os.getenv('FEATURE_NEW_ALGORITHM', 'false').lower(
            'enable_experimental_feature': os.getenv('FEATURE_EXPERIMENTAL', 'false').
        }

    def is_enabled(self, flag_name: str) -> bool:
        return self.flags.get(flag_name, False)

# Usage
if feature_flags.is_enabled('enable_new_algorithm'):
    result = await new_algorithm(data)
else:
    result = await legacy_algorithm(data)
```

Conclusion

This System Integration Guide provides the foundation for building production-ready AI applications using the Enterprise AI Infrastructure platform. By following these patterns and best practices, you can:

1. **Seamlessly integrate** all platform components
2. **Monitor and optimize** performance across the entire system
3. **Deploy with confidence** using proven production patterns
4. **Scale efficiently** as your AI applications grow
5. **Troubleshoot effectively** when issues arise

Quick Start Checklist

- [ ] Set up unified configuration management
- [ ] Implement correlation tracking across all components
- [ ] Configure comprehensive logging and monitoring
- [ ] Set up health checks for all components
- [ ] Implement graceful shutdown handling
- [ ] Configure performance monitoring and alerting
- [ ] Set up automated testing for integration points
- [ ] Deploy using containerization (Docker/Kubernetes)
- [ ] Implement security best practices
- [ ] Set up performance optimization monitoring

Next Steps

1. **Start with a simple integration** using the document intelligence example
2. **Add monitoring and observability** early in your development process
3. **Implement comprehensive testing** for all integration points
4. **Gradually optimize performance** based on real usage patterns
5. **Scale horizontally** using the provided Kubernetes configurations

The Enterprise AI Infrastructure platform gives you everything you need to build world-class AI applications. Now go build the future!

This integration guide is part of the Enterprise AI Infrastructure project. For questions, issues, or contributions, please visit our GitHub repository.