**SutazAI Technical Blueprint - 13da399582**

**User:**

Write code for - # SutazAI: Virtualized AGI/ASI System Technical Blueprint  
  
## Executive Summary  
  
This document outlines the comprehensive technical blueprint and implementation roadmap for developing SutazAI, a fully autonomous Artificial General Intelligence (AGI) system capable of secure recursive self-improvement toward Artificial Superintelligence (ASI). The system is designed to operate entirely on a Dell PowerEdge R720 server as a self-contained entity with zero cloud dependencies or external API calls.  
  
SutazAI integrates software-based neural architectures with formally verified ethical constraints in a fully virtualized environment, achieving secure recursive self-improvement from a narrow AI seed to ASI emergence through a staged developmental protocol—all within the hardware constraints of the server.  
  
---  
  
## 1. System Architecture Overview  
  
### 1.1 Hardware Constraints & Virtualization Strategy  
  
#### Dell PowerEdge R720 Specifications  
- \*CPU\*: 12 × Intel® Xeon® E5-2640 @ 2.50GHz (24 threads total)  
- \*RAM\*: 127.94 GB  
- \*Storage\*: 14.31 TB  
- \*No dedicated GPU\* (CPU-only operation with potential for future GPU integration)  
  
#### Virtualization Strategy  
- All components will operate as software processes within the server  
- No physical circuits, hardware modifications, or external devices required  
- Resource allocation optimized for the limited CPU resources  
- Memory-efficient implementations to operate within 128GB RAM constraint  
- Disk I/O optimization to leverage available storage while minimizing wear  
  
#### Resource Management  
- Dynamic CPU core allocation based on workload priority  
- Memory management with garbage collection optimization  
- Disk cache strategy to minimize I/O operations  
- Process scheduling for optimal multi-threaded performance  
- Swap management to handle potential memory pressure  
  
### 1.2 Core Architecture Components  
  
#### Software-Based Neural Framework  
- \*Virtual Neural Networks\* implemented entirely in software  
- \*Dynamic Parameter Adjustment\* simulating synaptic plasticity  
- \*CPU-Optimized Attention Mechanisms\* with thread-level parallelism  
- \*Multimodal Processing Pipeline\* for text, code, images, and structured data  
  
#### Ethical Constraint System  
- \*Software-Implemented Decision Boundaries\* verified through formal methods  
- \*Runtime Constraint Monitoring\* integrated into the execution environment  
- \*Provably Aligned Reward Modeling\* with cryptographic integrity verification  
- \*Moral Utility Functions\* embedded within all decision processes  
  
#### Self-Modification Control Framework  
- \*Virtualized Sandbox Environments\* for testing code changes  
- \*Cryptographic Verification\* of all code modifications  
- \*Detailed Audit Logging\* for transparency and accountability  
- \*Version Control System\* with immutable history  
  
#### Server-Optimized Framework  
- \*CPU-Only Quantization Pipeline\* for model compression  
- \*Thread-Aware Task Distribution\* for parallel processing  
- \*Memory-Conscious Architecture\* to prevent swapping  
- \*I/O Reduction Techniques\* to minimize disk access  
  
#### Virtual Security Architecture  
- \*Software-Defined Isolation\* using containerization  
- \*Process-Level Security Controls\* with least privilege principles  
- \*Memory Protection\* using OS-level safeguards  
- \*Digital Signing\* for executable integrity  
  
### 1.3 System Integration Topology  
  
The SutazAI system integrates these components in a layered architecture:  
  
1. \*Hardware Layer\*: Dell PowerEdge R720 server (12× Xeon E5-2640, 128GB RAM, 14TB storage)  
2. \*OS & Runtime Layer\*: Ubuntu Server with optimized kernel parameters  
3. \*Model Management Layer\*: Quantized LLMs with dynamic loading/unloading  
4. \*Agent Orchestration Layer\*: Lightweight agent framework with minimal overhead  
5. \*Application Layer\*: FastAPI backend and Streamlit web UI for human interaction  
6. \*Monitoring Layer\*: Resource usage tracking and optimization  
  
---  
  
## 2. Hardware-Aware AGI/ASI System Components  
  
### 2.1 CPU-Optimized Neural Architecture  
  
#### Efficient Neural Network Implementation  
- Vector instruction optimization (AVX2) for neural computations  
- Batch processing to maximize CPU cache utilization  
- Thread pool management for parallel neural operations  
- Fixed-point arithmetic where possible to reduce computational load  
- Model quantization (8-bit and 4-bit) to reduce memory footprint  
  
python  
# Conceptual implementation of CPU-optimized neural computation  
class CPUOptimizedTransformer:  
 def \_init\_(self, model\_path, quantize\_bits=8):  
 self.model = self.\_load\_quantized\_model(model\_path, quantize\_bits)  
 self.thread\_pool = ThreadPoolExecutor(max\_workers=os.cpu\_count())  
   
 def \_load\_quantized\_model(self, path, bits):  
 # Load model with quantization to reduce memory footprint  
 if bits == 8:  
 return AutoModelForCausalLM.from\_pretrained(  
 path,   
 load\_in\_8bit=True,  
 device\_map="auto",  
 torch\_dtype=torch.float16  
 )  
 elif bits == 4:  
 return AutoModelForCausalLM.from\_pretrained(  
 path,   
 load\_in\_4bit=True,  
 device\_map="auto",  
 torch\_dtype=torch.float16  
 )  
   
 def generate(self, prompt, max\_tokens=512):  
 # Optimize batch size for CPU cache utilization  
 optimal\_batch\_size = self.\_determine\_optimal\_batch\_size()  
   
 # Use thread pool for parallel processing where possible  
 return self.thread\_pool.submit(  
 self.\_generate\_with\_batching,   
 prompt,   
 max\_tokens,   
 optimal\_batch\_size  
 ).result()  
  
  
#### Memory-Efficient Multimodal Processing  
- Dynamic model loading/unloading to manage memory usage  
- Streaming processing for large inputs to avoid memory spikes  
- Progressive model cascade (small models first, larger only when needed)  
- Shared embedding spaces across modalities to reduce duplication  
- Memory-mapped file access for large datasets  
  
#### Distributed Computation Management  
- Workload distribution across available CPU cores  
- Priority-based scheduling for critical tasks  
- Asynchronous processing for non-blocking operations  
- Resource monitoring and adaptive task allocation  
- Throttling mechanisms to prevent CPU saturation  
  
### 2.2 Software-Based Ethical Framework  
  
#### Virtual Constraint System  
- Rule-based constraint verification using formal logic  
- Runtime policy enforcement through process monitoring  
- Decision validation through multi-level checking  
- Anomaly detection for unexpected behaviors  
- Automated reasoning for ethical decision-making  
  
python  
# Conceptual implementation of software-based ethical constraint checking  
class VirtualEthicalConstraints:  
 def \_init\_(self, constraints\_path):  
 self.constraints = self.\_load\_constraints(constraints\_path)  
 self.verification\_engine = self.\_initialize\_verifier()  
 self.decision\_log = []  
   
 def verify\_action(self, proposed\_action, context):  
 # Check if action violates any constraints  
 for constraint in self.constraints:  
 verification\_result = self.verification\_engine.verify(  
 constraint=constraint,  
 action=proposed\_action,  
 context=context  
 )  
   
 if not verification\_result.compliant:  
 self.decision\_log.append({  
 "timestamp": time.time(),  
 "action": proposed\_action,  
 "constraint": constraint,  
 "result": "rejected",  
 "reason": verification\_result.reason  
 })  
 return False, verification\_result.reason  
   
 # Action passed all constraints  
 self.decision\_log.append({  
 "timestamp": time.time(),  
 "action": proposed\_action,  
 "result": "approved"  
 })  
 return True, "Action complies with all ethical constraints"  
  
  
#### Digital Integrity Verification  
- Cryptographic hashing for process verification  
- Digital signatures for authenticating system changes  
- Tamper-evident logging using hash chains  
- State verification through cryptographic proofs  
- Integrity checking for critical components  
  
### 2.3 Virtualized Self-Modification System  
  
#### Software Sandbox Environment  
- Virtual execution environments for testing modifications  
- Process isolation for secure evaluation  
- Snapshot-based state management for rollback  
- Resource limiting to prevent denial of service  
- Behavioral analysis for safety verification  
  
python  
# Conceptual implementation of virtualized self-modification control  
class VirtualSandboxController:  
 def \_init\_(self, base\_system\_path):  
 self.base\_system\_path = base\_system\_path  
 self.active\_sandboxes = {}  
   
 def create\_sandbox(self, modification\_id):  
 # Create a virtual environment for testing modifications  
 sandbox\_id = f"sandbox\_{modification\_id}\_{int(time.time())}"  
 sandbox\_path = f"/tmp/sutazai\_sandboxes/{sandbox\_id}"  
   
 # Clone the base system into the sandbox  
 shutil.copytree(self.base\_system\_path, sandbox\_path)  
   
 # Initialize resource limits  
 resource\_limits = {  
 "cpu\_percent": 50,  
 "memory\_mb": 16000,  
 "disk\_mb": 5000,  
 "time\_seconds":: 600 # 10 minute timeout  
 }  
   
 self.active\_sandboxes[sandbox\_id] = {  
 "path": sandbox\_path,  
 "status": "initialized",  
 "resource\_limits": resource\_limits,  
 "creation\_time": time.time()  
 }  
   
 return sandbox\_id  
   
 def apply\_modification(self, sandbox\_id, modification\_script):  
 # Apply a proposed modification in the sandbox  
 if sandbox\_id not in self.active\_sandboxes:  
 return False, "Sandbox not found"  
   
 sandbox = self.active\_sandboxes[sandbox\_id]  
 sandbox["status"] = "applying\_modification"  
   
 # Execute the modification in the sandbox with resource limits  
 result = self.\_run\_with\_resource\_limits(  
 sandbox\_id,  
 modification\_script  
 )  
   
 sandbox["status"] = "modification\_applied"  
 sandbox["modification\_result"] = result  
   
 return True, "Modification applied in sandbox"  
   
 def verify\_sandbox(self, sandbox\_id, verification\_tests):  
 # Run verification tests in the sandbox  
 if sandbox\_id not in self.active\_sandboxes:  
 return False, "Sandbox not found"  
   
 sandbox = self.active\_sandboxes[sandbox\_id]  
 sandbox["status"] = "verification"  
   
 # Execute verification tests  
 verification\_results = self.\_run\_verification\_tests(  
 sandbox\_id,  
 verification\_tests  
 )  
   
 sandbox["status"] = "verified"  
 sandbox["verification\_results"] = verification\_results  
   
 return verification\_results  
  
  
#### Modification Approval Process  
- Multi-stage verification before implementation  
- Rollback capability for failed modifications  
- Performance impact analysis for proposed changes  
- Compatibility validation with existing components  
- Incremental deployment of verified changes  
  
### 2.4 CPU-Aware Optimization Framework  
  
#### Thread-Optimized Processing  
- Thread affinity management for cache efficiency  
- Load balancing across available cores  
- Thread pool optimization for various workloads  
- Priority-based thread scheduling  
- Minimizing context switching overhead  
  
#### Memory Optimization  
- Shared memory utilization for inter-process communication  
- Memory pooling to reduce allocation overhead  
- Cache-friendly data structures and algorithms  
- Garbage collection tuning for predictable performance  
- Compressed in-memory data structures  
  
python  
# Conceptual implementation of memory-optimized inference  
class MemoryEfficientInference:  
 def \_init\_(self, model\_manager):  
 self.model\_manager = model\_manager  
 self.active\_models = {}  
 self.memory\_monitor = MemoryMonitor()  
   
 def run\_inference(self, model\_name, input\_data, priority="normal"):  
 # Check if we have enough memory to load the model  
 required\_memory = self.model\_manager.get\_model\_memory\_requirement(model\_name)  
 available\_memory = self.memory\_monitor.get\_available\_memory()  
   
 if required\_memory > available\_memory:  
 # Not enough memory - unload less important models  
 self.\_free\_memory\_for\_model(required\_memory)  
   
 # Load model if not already loaded  
 if model\_name not in self.active\_models:  
 self.active\_models[model\_name] = self.model\_manager.load\_model(  
 model\_name,   
 quantize=True  
 )  
   
 # Run inference with appropriate priority  
 result = self.\_prioritized\_inference(  
 model\_name,  
 input\_data,  
 priority  
 )  
   
 # Consider unloading model based on memory pressure  
 self.\_consider\_unloading\_models()  
   
 return result  
   
 def \_free\_memory\_for\_model(self, required\_memory):  
 # Unload models based on priority and last used time  
 models\_to\_unload = []  
   
 for model\_name, model\_info in self.active\_models.items():  
 if model\_info["priority"] == "low" and not model\_info["in\_use"]:  
 models\_to\_unload.append(model\_name)  
   
 # Unload models until we have enough memory  
 for model\_name in models\_to\_unload:  
 self.model\_manager.unload\_model(model\_name)  
 del self.active\_models[model\_name]  
   
 # Check if we now have enough memory  
 if self.memory\_monitor.get\_available\_memory() >= required\_memory:  
 break  
  
  
#### I/O Optimization  
- Buffered I/O operations to reduce disk access  
- Database query optimization for efficient data retrieval  
- Incremental data processing for large datasets  
- Memory-mapped files for large dataset access  
- Asynchronous I/O for non-blocking operations  
  
### 2.5 Software-Defined Security Framework  
  
#### Process Isolation  
- Linux namespaces for process isolation  
- Control groups (cgroups) for resource limitation  
- Capability-based permission model  
- Process monitoring for anomaly detection  
- Secure inter-process communication channels  
  
#### Access Control  
- Principle of least privilege for all processes  
- Role-based access control for administrative functions  
- File system permission hardening  
- Memory protection using OS mechanisms  
- Network interface control through software firewalls  
  
python  
# Conceptual implementation of process-based security controls  
class ProcessSecurityManager:  
 def \_init\_(self):  
 self.secured\_processes = {}  
   
 def launch\_secured\_process(self, executable\_path, args, security\_profile):  
 # Create a secure environment for the process  
 process\_id = str(uuid.uuid4())  
   
 # Define security controls based on profile  
 namespace\_config = self.\_get\_namespace\_config(security\_profile)  
 cgroup\_config = self.\_get\_cgroup\_config(security\_profile)  
 capability\_config = self.\_get\_capability\_config(security\_profile)  
   
 # Launch process with security controls  
 process = subprocess.Popen(  
 ["systemd-run", "--scope", "--unit=sutazai-" + process\_id] +  
 namespace\_config +  
 cgroup\_config +  
 capability\_config +  
 [executable\_path] + args,  
 stdout=subprocess.PIPE,  
 stderr=subprocess.PIPE  
 )  
   
 self.secured\_processes[process\_id] = {  
 "process": process,  
 "executable": executable\_path,  
 "security\_profile": security\_profile,  
 "start\_time": time.time()  
 }  
   
 return process\_id  
   
 def monitor\_process(self, process\_id):  
 # Get process stats and check for anomalies  
 if process\_id not in self.secured\_processes:  
 return None  
   
 process\_info = self.secured\_processes[process\_id]  
   
 # Get resource usage  
 resource\_usage = self.\_get\_process\_resources(process\_id)  
   
 # Check for anomalies  
 anomalies = self.\_detect\_anomalies(process\_id, resource\_usage)  
   
 return {  
 "process\_id": process\_id,  
 "status": "running" if process\_info["process"].poll() is None else "terminated",  
 "resource\_usage": resource\_usage,  
 "anomalies": anomalies  
 }  
  
  
#### Cryptographic Protection  
- Data encryption for sensitive information  
- Secure key management in software  
- Cryptographic validation of processes  
- Digital signatures for code authentication  
- Secure pseudo-random number generation  
  
---  
  
## 3. Implementation Roadmap (Hardware-Aware)  
  
### 3.1 Phase 1: Core System Setup (Months 1-6)  
  
#### Milestone 1.1: Server Preparation & Baseline (Month 1)  
- Hardware diagnostics and performance benchmarking  
- OS installation with security hardening  
- Kernel parameter optimization for AI workloads  
- Memory and disk I/O tuning  
- Network interface configuration (disabled where possible)  
  
#### Milestone 1.2: Efficient Model Foundation (Months 2-3)  
- Model evaluation and selection based on CPU performance  
- Implementation of dynamic model loading/unloading framework  
- Vector database setup with memory optimization  
- Model quantization pipeline for 8-bit and 4-bit precision  
- Thread-optimized inference engine  
  
#### Milestone 1.3: Resource-Aware Framework (Months 3-4)  
- Implementation of resource monitoring system  
- Dynamic resource allocation mechanisms  
- Thread pool management system  
- Memory pressure handling algorithms  
- I/O optimization framework  
  
#### Milestone 1.4: Agent System Foundations (Months 4-6)  
- Lightweight agent framework implementation  
- Inter-agent communication protocols  
- Resource-efficient task planning system  
- Tool integration framework  
- Agent orchestration engine  
  
### 3.2 Phase 2: Core Capabilities (Months 7-12)  
  
#### Milestone 2.1: Document Processing System (Months 7-8)  
- Memory-efficient document parsing  
- Streaming-based document processing  
- Incremental knowledge extraction  
- Efficient vector embedding generation  
- Memory-mapped document storage  
  
#### Milestone 2.2: Code Processing Framework (Months 8-10)  
- Code analysis tools with memory optimization  
- CPU-efficient code generation  
- Incremental code analysis framework  
- Multi-threaded code verification  
- Memory-efficient code indexing  
  
#### Milestone 2.3: Learning System (Months 10-11)  
- Parameter-efficient fine-tuning techniques  
- Memory-conscious learning algorithms  
- Incremental model updates  
- Training data compression techniques  
- Distributed learning across CPU cores  
  
#### Milestone 2.4: Web UI & API (Months 11-12)  
- Lightweight FastAPI backend implementation  
- Resource-efficient Streamlit UI  
- Asynchronous request handling  
- Connection pooling for database access  
- Caching strategies for frequently accessed data  
  
### 3.3 Phase 3: Advanced Capabilities (Months 13-18)  
  
#### Milestone 3.1: Self-Optimization Framework (Months 13-14)  
- System performance profiling  
- Bottleneck identification  
- Resource-aware optimization strategies  
- Code path optimization  
- Memory usage optimization  
  
#### Milestone 3.2: Enhanced Reasoning (Months 14-16)  
- Multi-step reasoning with memory optimization  
- Efficient knowledge retrieval mechanisms  
- Cached reasoning for similar queries  
- Memory-efficient logical inference  
- Resource-aware planning algorithms  
  
#### Milestone 3.3: Multi-Agent Orchestration (Months 16-18)  
- Efficient agent communication protocols  
- Resource allocation across multiple agents  
- Task prioritization and scheduling  
- Memory sharing between agents  
- Load balancing for agent workloads  
  
### 3.4 Phase 4: Self-Improvement & ASI Capabilities (Months 19-24)  
  
#### Milestone 4.1: Bounded Self-Improvement (Months 19-20)  
- Safe code modification framework  
- Test-driven improvement mechanisms  
- Performance-focused optimization  
- Resource usage improvement strategies  
- Incremental capability enhancement  
  
#### Milestone 4.2: Advanced Self-Understanding (Months 20-22)  
- System architecture self-analysis  
- Performance self-monitoring  
- Resource usage optimization  
- Code quality self-improvement  
- Algorithm selection optimization  
  
#### Milestone 4.3: ASI Transition Framework (Months 22-24)  
- Capability expansion monitoring  
- Resource-aware growth strategies  
- Bounded self-enhancement mechanisms  
- Memory-efficient cognitive expansion  
- CPU-optimized reasoning systems  
  
---  
  
## 4. Virtualized System Integration  
  
### 4.1 Software-Defined Component Integration  
  
#### Process Management  
- Systemd service definitions for component management  
- Docker/Podman containers for isolated components (optional)  
- Process supervision for automatic restarts  
- Init scripts for controlled startup/shutdown  
- Resource limits for each component  
  
#### Inter-Process Communication  
- REST APIs for synchronous communication  
- Message queues for asynchronous communication  
- Shared memory for high-performance data exchange  
- Unix sockets for local communication  
- File-based communication for simple interfaces  
  
#### Resource Orchestration  
- Dynamic CPU allocation based on workload  
- Memory allocation with pressure monitoring  
- Disk space management and cleanup  
- Network resource management  
- I/O scheduling optimization  
  
### 4.2 Software Dependency Management  
  
#### Package Management  
- Conda/Mamba environments for Python dependencies  
- Containerized dependencies for isolation  
- Virtual environments for clean dependency trees  
- Local package cache for offline operation  
- Version pinning for stability  
  
#### Model Management  
- Efficient model storage and versioning  
- Compressed model formats for space efficiency  
- Lazy loading of model components  
- Shared model layers where applicable  
- Model pruning for size reduction  
  
#### Data Storage  
- Efficient database schema design  
- Indexing strategies for fast retrieval  
- Data compression for storage efficiency  
- Data lifecycle management  
  
### 4.3 Memory-Optimized Development Environment  
  
#### Code Management  
- Git-based version control with shallow clones  
- Incremental builds for faster development  
- Code optimization for memory efficiency  
- Compiler optimization for performance  
- Link-time optimization for executable size  
  
#### Testing Framework  
- Memory-efficient unit testing  
- Performance regression testing  
- Resource usage monitoring during tests  
- Stress testing under memory constraints  
- Long-running stability tests  
  
#### Documentation System  
- Lightweight documentation generation  
- On-demand documentation rendering  
- Search optimization for documentation access  
- Memory-efficient knowledge representation  
- Progressive loading of documentation content  
  
---  
  
## 5. CPU-Optimized Web UI and Backend System  
  
### 5.1 FastAPI Backend (Resource-Efficient)  
  
#### API Design  
- Lightweight endpoint design  
- Asynchronous request handling  
- Batched operations for efficiency  
- Request throttling for load management  
- Connection pooling for database access  
  
python  
# FastAPI backend with resource-efficient design  
from fastapi import FastAPI, BackgroundTasks, Depends, HTTPException  
from fastapi.middleware.cors import CORSMiddleware  
import asyncio  
import psutil  
import logging  
from datetime import datetime  
from typing import List, Dict, Any, Optional  
  
# Setup logging to file to avoid console I/O overhead  
logging.basicConfig(  
 level=logging.INFO,  
 filename="/opt/sutazaiapp/logs/backend.log",  
 format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'  
)  
logger = logging.getLogger("sutazai-backend")  
  
app = FastAPI(title="SutazAI Backend")  
  
# Limit CORS to reduce overhead  
app.add\_middleware(  
 CORSMiddleware,  
 allow\_origins=["http://localhost:8501"], # Streamlit UI only  
 allow\_credentials=True,  
 allow\_methods=["\*"],  
 allow\_headers=["\*"],  
)  
  
# Connection pool for database  
db\_pool = None  
# Thread pool for CPU-bound tasks  
thread\_pool = None  
  
@app.on\_event("startup")  
async def startup\_event():  
 global db\_pool, thread\_pool  
 # Initialize connection pool with size based on available resources  
 cpu\_count = psutil.cpu\_count()  
 max\_connections = min(cpu\_count \* 2, 20) # Cap at 20 connections  
   
 # Initialize database connection pool  
 db\_pool = await init\_db\_pool(max\_connections=max\_connections)  
   
 # Initialize thread pool for CPU-bound tasks  
 thread\_pool = ThreadPoolExecutor(max\_workers=cpu\_count - 1)  
   
 logger.info(f"Backend started with {max\_connections} db connections and {cpu\_count-1} worker threads")  
  
@app.get("/system/status")  
async def get\_system\_status():  
 """Get system status with minimal resource usage"""  
 # Use cached values for CPU/memory that update every 5 seconds  
 return {  
 "status": "Running",  
 "cpu\_usage": get\_cached\_cpu\_usage(),  
 "memory\_usage": get\_cached\_memory\_usage(),  
 "components": get\_cached\_component\_status()  
 }  
  
@app.post("/inference")  
async def run\_inference(request: InferenceRequest, background\_tasks: BackgroundTasks):  
 """Run inference with resource management"""  
 # Check available memory before accepting large requests  
 if not has\_sufficient\_memory(request):  
 raise HTTPException(status\_code=503, detail="Insufficient system resources")  
   
 # For long-running inference, use background task  
 if request.expected\_duration > 5:  
 task\_id = generate\_task\_id()  
 background\_tasks.add\_task(run\_inference\_task, request, task\_id)  
 return {"task\_id": task\_id, "status": "processing"}  
   
 # For quick inference, run directly  
 return await run\_inference\_sync(request)  
  
  
#### Resource Management  
- Memory usage monitoring  
- Connection pooling for databases  
- Request queuing for high load periods  
- Background task processing for long operations  
- Cached responses for frequent requests  
  
#### API Optimization  
- Response compression  
- Request batching  
- Pagination for large responses  
- Field filtering to reduce response size  
- Conditional requests with ETags  
  
### 5.2 Streamlit UI (Lightweight)  
  
#### Efficient UI Design  
- Progressive loading of components  
- Minimal state management  
- Efficient data serialization  
- Lazy loading of UI elements  
- Cached rendering where possible  
  
python  
# Streamlit UI with optimized resource usage  
import streamlit as st  
import requests  
import json  
import time  
import os  
from datetime import datetime  
  
# Reduce Streamlit's memory footprint  
st.set\_page\_config(  
 page\_title="SutazAI Management",  
 layout="wide",  
 initial\_sidebar\_state="collapsed" # Save memory by starting collapsed  
)  
  
# Use session state sparingly to reduce memory usage  
if "authenticated" not in st.session\_state:  
 st.session\_state.authenticated = False  
if "last\_update" not in st.session\_state:  
 st.session\_state.last\_update = 0  
if "system\_status" not in st.session\_state:  
 st.session\_state.system\_status = None  
  
# Function to efficiently fetch data with caching  
def get\_system\_status(force\_refresh=False):  
 current\_time = time.time()  
 # Only refresh data every 10 seconds unless forced  
 if force\_refresh or (current\_time - st.session\_state.last\_update) > 10:  
 try:  
 response = requests.get("http://localhost:8000/system/status", timeout=2)  
 st.session\_state.system\_status = response.json()  
 st.session\_state.last\_update = current\_time  
 except Exception as e:  
 # Don't update timestamp on error to allow retry  
 if st.session\_state.system\_status is None:  
 st.session\_state.system\_status = {"status": "Error", "error": str(e)}  
   
 return st.session\_state.system\_status  
  
# Main application with optimized rendering  
def main():  
 # Authentication with minimal overhead  
 if not st.session\_state.authenticated:  
 st.title("SutazAI Login")  
 col1, col2 = st.columns(2)  
 with col1:  
 username = st.text\_input("Username")  
 with col2:  
 password = st.text\_input("Password", type="password")  
   
 if st.button("Login"):  
 # Simple auth for demo - replace with secure auth  
 if username == "admin" and password == "sutazai":  
 st.session\_state.authenticated = True  
 st.experimental\_rerun()  
 else:  
 st.error("Invalid credentials")  
 return  
   
 # Main UI with tabbed interface to reduce rendering overhead  
 st.title("SutazAI Management Console")  
 tab1, tab2, tab3 = st.tabs(["System", "Models", "Agents"])  
   
 # System tab - load efficiently  
 with tab1:  
 status = get\_system\_status()  
   
 # Refresh button - only re-fetch data when clicked  
 if st.button("Refresh Status"):  
 status = get\_system\_status(force\_refresh=True)  
   
 # Display system metrics with minimal components  
 col1, col2, col3 = st.columns(3)  
 col1.metric("Status", status.get("status", "Unknown"))  
 col2.metric("CPU Usage", f"{status.get('cpu\_usage', 0)}%")  
 col3.metric("Memory Usage", f"{status.get('memory\_usage', 0)}%")  
   
 # Only display components if we have them  
 if "components" in status:  
 st.subheader("Components")  
 # Use native table instead of DataFrame for efficiency  
 st.table(status["components"])  
  
  
#### Data Visualization  
- Simplified visualizations for resource efficiency  
- Data aggregation before visualization  
- Client-side data processing where appropriate  
- Incremental rendering for large datasets  
- Downsampling for time-series data  
  
#### User Experience  
- Minimal animations and effects  
- Efficient state management  
- Optimized asset loading  
- Responsive design for different screens  
- Performance-focused implementation  
  
### 5.3 Database and Storage  
  
#### Efficient Data Storage  
- Optimized database schema design  
- Indexing strategies for query performance  
- Denormalization where appropriate for read performance  
- Document compression for storage efficiency  
- Connection pooling for database access  
  
#### Query Optimization  
- Prepared statements for frequent queries  
- Query caching for repeated requests  
- Pagination for large result sets  
- Optimized joins and aggregations  
- Execution plan optimization  
  
#### Vector Storage Optimization  
- Dimensionality reduction for embeddings  
- Quantized vector storage  
- Chunked vector retrieval  
- Fast approximate nearest neighbor search  
- Cached similarity calculations  
  
---  
  
## 6. Deployment and Management  
  
### 6.1 Resource-Efficient Deployment Scripts  
  
#### System Initialization  
bash  
#!/bin/bash  
# deploy.sh - Optimized deployment script for SutazAI  
  
# Set up logging with minimal I/O  
LOG\_DIR="/opt/sutazaiapp/logs"  
MAIN\_LOG="$LOG\_DIR/sutazai\_deploy.log"  
mkdir -p $LOG\_DIR  
  
# Log function that minimizes disk I/O with buffering  
log() {  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" >> $MAIN\_LOG  
 # Only echo to console for important messages  
 if [[ "$2" == "important" ]]; then  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"  
 fi  
}  
  
# Error handling  
set -e  
trap 'echo "Error occurred at line $LINENO. Deployment failed."; exit 1' ERR  
  
log "Starting SutazAI system deployment..." "important"  
  
# Create necessary directories efficiently (create parent directories with one command)  
log "Creating directory structure..."  
mkdir -p \  
 /opt/sutazaiapp/backend \  
 /opt/sutazaiapp/ui \  
 /opt/sutazaiapp/vector\_db \  
 /opt/sutazaiapp/agents \  
 /opt/sutazaiapp/models \  
 /opt/sutazaiapp/logs \  
 /opt/sutazaiapp/data  
  
# Setup Python virtual environment if not exists  
if [ ! -d "/opt/sutazaiapp/venv-sutazaiapp" ]; then  
 log "Creating Python virtual environment..."  
 # Use system site packages to reduce duplication of packages  
 python3 -m venv /opt/sutazaiapp/venv-sutazaiapp --system-site-packages  
fi  
  
# Activate virtual environment  
source /opt/sutazaiapp/venv-sutazaiapp/bin/activate  
log "Virtual environment activated"  
  
# Install required packages efficiently  
log "Installing Python dependencies..."  
# Use pip's dependency resolver more efficiently  
pip install --no-cache-dir --upgrade pip  
  
# Install packages in one batch to reduce solver runs  
log "Installing main packages..."  
pip install --no-cache-dir \  
 fastapi uvicorn streamlit chromadb \  
 faiss-cpu langchain pydantic \  
 python-dotenv psutil  
  
# Optimize models directory  
log "Setting up models directory..."  
mkdir -p /opt/sutazaiapp/models/cache  
  
# Create system control scripts  
log "Creating system control scripts..." "important"  
  
# Create start.sh with resource-aware settings  
cat > /opt/sutazaiapp/start.sh << 'EOF'  
#!/bin/bash  
# start.sh - Resource-optimized script to start SutazAI components  
  
# Set up logging  
LOG\_DIR="/opt/sutazaiapp/logs"  
MAIN\_LOG="$LOG\_DIR/sutazai\_start.log"  
mkdir -p $LOG\_DIR  
  
# Log function  
log() {  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" >> $MAIN\_LOG  
 # Only echo important messages to console  
 if [[ "$2" == "important" ]]; then  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"  
 fi  
}  
  
# Activate virtual environment  
source /opt/sutazaiapp/venv-sutazaiapp/bin/activate  
log "Virtual environment activated"  
  
# Get available resources  
TOTAL\_MEM=$(free -m | awk '/^Mem:/{print $2}')  
AVAIL\_MEM=$(free -m | awk '/^Mem:/{print $7}')  
CPU\_CORES=$(nproc)  
  
log "System has $CPU\_CORES CPU cores and $AVAIL\_MEM MB available memory" "important"  
  
# Adjust server resources based on available memory  
# Reserve at least 2GB for the OS  
BACKEND\_WORKERS=$(( CPU\_CORES > 4 ? 4 : CPU\_CORES ))  
  
# Start FastAPI backend with resource constraints  
log "Starting FastAPI backend with $BACKEND\_WORKERS workers..." "important"  
cd /opt/sutazaiapp/backend  
nohup uvicorn main:app --host 0.0.0.0 --port 8000 --workers $BACKEND\_WORKERS > $LOG\_DIR/backend.log 2>&1 &  
BACKEND\_PID=$!  
log "FastAPI backend started with PID: $BACKEND\_PID"  
  
# Start ChromaDB with memory limits  
log "Starting ChromaDB vector database..." "important"  
cd /opt/sutazaiapp/vector\_db  
CHROMA\_MEMORY\_LIMIT=$(( AVAIL\_MEM / 4 )) # Limit to 25% of available memory  
export CHROMA\_MEMORY\_LIMIT\_MB=$CHROMA\_MEMORY\_LIMIT  
nohup python -m chromadb.server --host 0.0.0.0 --port 8100 > $LOG\_DIR/chromadb.log 2>&1 &  
CHROMA\_PID=$!  
log "ChromaDB started with PID: $CHROMA\_PID and $CHROMA\_MEMORY\_LIMIT MB memory limit"  
  
# Start Streamlit UI with resource constraints  
log "Starting Streamlit UI..." "important"  
cd /opt/sutazaiapp/ui  
nohup streamlit run app.py --server.port 8501 --browser.serverAddress localhost --server.maxUploadSize 10 > $LOG\_DIR/ui.log 2>&1 &  
UI\_PID=$!  
log "Streamlit UI started with PID: $UI\_PID"  
  
# Start AI Agents if sufficient memory available  
REMAINING\_MEM=$(free -m | awk '/^Mem:/{print $7}')  
if [ $REMAINING\_MEM -gt 4000 ]; then # If more than 4GB still available  
 log "Starting AI Agents..." "important"  
 cd /opt/sutazaiapp/agents  
 nohup python agent\_manager.py > $LOG\_DIR/agents.log 2>&1 &  
 AGENTS\_PID=$!  
 log "AI Agents started with PID: $AGENTS\_PID"  
else  
 log "Insufficient memory to start AI Agents. Please free up resources." "important"  
fi  
  
# Verify services are running  
sleep 5  
log "Verifying services..." "important"  
  
# Create status file for monitoring  
cat > /opt/sutazaiapp/status.json << EOSTATUS  
{  
 "last\_update": "$(date '+%Y-%m-%d %H:%M:%S')",  
 "components": [  
 {"name": "FastAPI Backend", "pid": $BACKEND\_PID, "port": 8000},  
 {"name": "ChromaDB", "pid": $CHROMA\_PID, "port": 8100},  
 {"name": "Streamlit UI", "pid": $UI\_PID, "port": 8501}  
 ],  
 "resources": {  
 "total\_memory\_mb": $TOTAL\_MEM,  
 "available\_memory\_mb": $AVAIL\_MEM,  
 "cpu\_cores": $CPU\_CORES  
 }  
}  
EOSTATUS  
  
log "SutazAI system startup completed" "important"  
echo "SutazAI system has been started. Access the UI at http://localhost:8501"  
EOF  
  
# Make scripts executable  
chmod +x /opt/sutazaiapp/start.sh  
  
log "SutazAI system deployment completed successfully" "important"  
echo "SutazAI system has been deployed. Run './start.sh' to start the system."  
  
  
#### Component Management  
- Systemd service definitions for component management  
- Resource limits in service configurations  
- Start/stop scripts with proper sequencing  
- Status monitoring scripts  
- Log rotation configuration  
  
#### Configuration Management  
- Environment-based configuration  
- Configuration validation  
- Sensible defaults for all settings  
- Dynamic configuration reloading  
  
### 6.2 Resource Monitoring and Optimization  
  
#### System Monitoring  
- CPU usage tracking and optimization  
- Memory usage monitoring  
- Disk I/O performance monitoring  
- Process resource utilization tracking  
- Component health checks  
  
#### Performance Optimization  
- Resource usage analysis  
- Bottleneck identification  
- Dynamic resource allocation  
- Performance tuning recommendations  
- Automated optimization actions  
  
#### Alerting and Reporting  
- Resource threshold alerts  
- Performance degradation detection  
- Component failure notifications  
- Regular performance reports  
- Trend analysis for resource usage  
  
### 6.3 Failure Recovery  
  
#### Automatic Recovery  
- Service restart on failure  
- Crash detection and remediation  
- Database recovery procedures  
- State synchronization after failures  
- Graceful degradation under resource pressure  
  
  
#### Logging and Debugging  
- Centralized logging with log rotation  
- Log level adjustment for debug information  
- Performance logging for optimization  
- Error tracking and aggregation  
- Root cause analysis tools  
  
---  
  
## 7. Safety and Verification  
  
### 7.1 Software-Based Safety Mechanisms  
  
#### Process Monitoring  
- Resource usage limits enforcement  
- Abnormal behavior detection  
- Runaway process termination  
- Performance degradation alerts  
- Process restart on failure  
  
#### Input Validation  
- Request sanitization  
- Parameter bounds checking  
- Rate limiting for API requests  
- Input size restrictions  
- Malformed request detection  
  
#### Output Verification  
- Response validation against constraints  
- Ethical output filtering  
- Content safety checking  
- Performance impact analysis  
- Response size limiting  
  
### 7.2 Verification Systems  
  
#### Automated Testing  
- Unit tests for components  
- Integration tests for system behavior  
- Performance regression tests  
- Resource usage verification  
- Security vulnerability testing  
  
#### Continuous Monitoring  
- Real-time system performance monitoring  
- Component health checking  
- Error rate tracking  
- Response time monitoring  
- Resource utilization tracking  
  
#### Formal Methods  
- Lightweight formal verification  
- Invariant checking for critical properties  
- Protocol verification for communications  
- Security property verification  
- Safety constraint validation  
  
### 7.3 Graceful Degradation  
  
#### Resource Prioritization  
- Critical service prioritization  
- Non-essential service reduction under load  
- Dynamic resource reallocation  
- Tiered service quality levels  
- Progressive feature disabling  
  
#### Fallback Mechanisms  
- Simpler model fallbacks when needed  
- Cached response usage when appropriate  
- Reduced complexity algorithms under load  
- Service quality adjustment  
- Default safe responses  
  
#### Recovery Procedures  
- Automated system recovery  
- Incremental service restoration  
- State reconciliation after failures  
- Prioritized component restart  
- User communication during degradation  
  
---  
  
## 8. Conclusion  
  
This technical blueprint outlines a comprehensive approach to implementing the SutazAI AGI/ASI system entirely through virtualized software components on a Dell PowerEdge R720 server. The design accounts for the server's hardware constraints (12× Xeon E5-2640, 128GB RAM, 14TB storage, no dedicated GPU) while still providing a path to achieve advanced AI capabilities.  
  
The implementation emphasizes:  
  
1. \*CPU optimization\* through efficient threading, memory management, and computational techniques  
2. \*Memory efficiency\* through model quantization, dynamic loading/unloading, and resource-aware algorithms  
3. \*Virtualized security\* using software-defined isolation, process controls, and cryptographic verification  
4. \*Scalable architecture\* that can expand capabilities while working within hardware constraints  
5. \*Fault tolerance\* through monitoring, automatic recovery, and graceful degradation  
  
By following this blueprint, the SutazAI system can be fully implemented as a virtualized set of software components, requiring no physical circuits or hardware modifications, while working efficiently within the server's resource constraints.  
  
The phased implementation approach ensures that foundational components are optimized before building more advanced capabilities, allowing the system to grow in sophistication while maintaining performance and stability on the available hardware.

**Abacus AI ChatLLM:**

Thought for 7 seconds

I need to create code that implements the technical blueprint for SutazAI, a system designed to function as a virtualized AGI/ASI on a Dell PowerEdge R720 server with specific hardware constraints. Let me analyze what the request is asking for and structure my response accordingly.  
  
The user is requesting me to write code that implements the SutazAI technical blueprint. This blueprint outlines a comprehensive system for developing an AGI/ASI that operates entirely on a Dell PowerEdge R720 server with no cloud dependencies. The document covers system architecture, implementation roadmap, various components, and deployment strategies.  
  
Given the extensive nature of the blueprint, I should focus on providing representative code samples for key components of the system, rather than attempting to implement the entire system. I'll identify the most critical aspects and provide well-structured, efficient code implementations for them.  
  
Let me focus on these key areas:  
  
1. CPU-Optimized Neural Architecture implementation  
2. Virtual Ethical Constraints system  
3. Self-Modification Sandbox Controller  
4. Memory-Efficient Inference Engine  
5. Process Security Manager  
6. FastAPI Backend (Resource-Efficient)  
7. Streamlit UI (Lightweight)  
8. Deployment Script  
  
I'll write code for these components that adheres to the requirements of being optimized for the hardware constraints of the Dell PowerEdge R720 server (12× Intel Xeon E5-2640, 128GB RAM, 14TB storage, no GPU).  
  
Let me start with a well-structured implementation of these components.

# SutazAI Implementation Core Components  
  
I'll focus on the most critical code components from the blueprint, optimized for the Dell PowerEdge R720 server constraints.  
  
## 1. CPU-Optimized Neural Architecture

# core/neural/cpu\_optimized\_transformer.py  
import os  
import numpy as np  
import torch  
from concurrent.futures import ThreadPoolExecutor  
from typing import Dict, List, Optional, Union  
from transformers import AutoModelForCausalLM, AutoTokenizer  
  
class CPUOptimizedTransformer:  
 def \_\_init\_\_(self, model\_path: str, quantize\_bits: int = 8):  
 """Initialize a CPU-optimized transformer model with quantization.  
  
 Args:  
 model\_path: Path to the model weights  
 quantize\_bits: Model quantization bit depth (4 or 8)  
 """  
 self.model\_path = model\_path  
 self.quantize\_bits = quantize\_bits  
 self.model = None  
 self.tokenizer = None  
 self.thread\_pool = ThreadPoolExecutor(max\_workers=os.cpu\_count())  
 self.\_load\_model()  
  
 def \_load\_model(self):  
 """Load the model with appropriate quantization."""  
 load\_kwargs = {  
 "device\_map": "auto",  
 "torch\_dtype": torch.float16,  
 }  
  
 if self.quantize\_bits == 8:  
 load\_kwargs["load\_in\_8bit"] = True  
 elif self.quantize\_bits == 4:  
 load\_kwargs["load\_in\_4bit"] = True  
 load\_kwargs["bnb\_4bit\_compute\_dtype"] = torch.float16  
 load\_kwargs["bnb\_4bit\_use\_double\_quant"] = True  
  
 self.model = AutoModelForCausalLM.from\_pretrained(  
 self.model\_path,  
 \*\*load\_kwargs  
 )  
 self.tokenizer = AutoTokenizer.from\_pretrained(self.model\_path)  
  
 def \_determine\_optimal\_batch\_size(self) -> int:  
 """Determine optimal batch size based on CPU cache and memory."""  
 # Simple heuristic based on available CPU cores and memory  
 cores = os.cpu\_count()  
 if cores <= 4:  
 return 1  
 elif cores <= 8:  
 return 2  
 else:  
 return 4  
  
 def generate(self, prompt: str, max\_tokens: int = 512) -> str:  
 """Generate text with optimized CPU usage  
  
 Args:  
 prompt: Input text prompt  
 max\_tokens: Maximum number of tokens to generate  
  
 Returns:  
 Generated text response  
 """  
 # Determine optimal batch size for CPU  
 batch\_size = self.\_determine\_optimal\_batch\_size()  
  
 # Submit to thread pool to avoid blocking  
 future = self.thread\_pool.submit(  
 self.\_generate\_with\_batching,  
 prompt,  
 max\_tokens,  
 batch\_size  
 )  
 return future.result()  
  
 def \_generate\_with\_batching(self, prompt: str, max\_tokens: int, batch\_size: int) -> str:  
 """Internal method to perform generation with batching  
  
 Args:  
 prompt: Input text prompt  
 max\_tokens: Maximum tokens to generate  
 batch\_size: Batch size for efficient CPU usage  
 """  
 # Tokenize input  
 input\_ids = self.tokenizer.encode(prompt, return\_tensors="pt")  
  
 # Generate with chunking for memory efficiency  
 generated\_tokens = []  
 chunk\_size = min(max\_tokens, 128) # Process in smaller chunks  
 remaining = max\_tokens  
  
 while remaining > 0:  
 current\_chunk = min(chunk\_size, remaining)  
 outputs = self.model.generate(  
 input\_ids,  
 max\_new\_tokens=current\_chunk,  
 do\_sample=True,  
 temperature=0.7,  
 batch\_size=batch\_size,  
 pad\_token\_id=self.tokenizer.eos\_token\_id  
 )  
  
 # Extract new tokens  
 new\_tokens = outputs[0][input\_ids.shape[1]:]  
 generated\_tokens.append(new\_tokens)  
  
 # Update for next iteration  
 input\_ids = outputs  
 remaining -= current\_chunk  
  
 # Combine results and decode  
 return self.tokenizer.decode(torch.cat(generated\_tokens), skip\_special\_tokens=True)

## 2. Virtual Ethical Constraints System

# core/ethics/constraint\_system.py  
import time  
import json  
import hashlib  
from typing import Dict, List, Tuple, Any  
from dataclasses import dataclass  
  
@dataclass  
class VerificationResult:  
 compliant: bool  
 reason: str = ""  
 details: Dict[str, Any] = None  
  
class VirtualEthicalConstraints:  
 def \_\_init\_\_(self, constraints\_path: str):  
 """Initialize the ethical constraint system  
  
 Args:  
 constraints\_path: Path to constraint definitions  
 """  
 self.constraints = self.\_load\_constraints(constraints\_path)  
 self.verification\_engine = self.\_initialize\_verifier()  
 self.decision\_log = []  
 self.log\_hash\_chain = self.\_initialize\_hash\_chain()  
  
 def \_load\_constraints(self, path: str) -> List[Dict]:  
 """Load constraint definitions from JSON file  
  
 Args:  
 path: Path to constraint file  
  
 Returns:  
 List of constraint definitions  
 """  
 with open(path, 'r') as f:  
 constraints = json.load(f)  
  
 # Validate constraint format  
 for constraint in constraints:  
 if not all(k in constraint for k in ['id', 'description', 'condition', 'severity']):  
 raise ValueError(f"Invalid constraint format: {constraint}")  
  
 return constraints  
  
 def \_initialize\_verifier(self):  
 """Initialize the verification engine  
  
 Returns:  
 Verification engine object  
 """  
 # This would be a more complex implementation in a real system  
 return ConstraintVerificationEngine()  
  
 def \_initialize\_hash\_chain(self) -> str:  
 """Initialize tamper-evident hash chain for logging  
  
 Returns:  
 Initial hash value  
 """  
 # Start with a hash of the constraints themselves for integrity  
 constraints\_str = json.dumps(self.constraints, sort\_keys=True)  
 return hashlib.sha256(constraints\_str.encode()).hexdigest()  
  
 def verify\_action(self, proposed\_action: Dict, context: Dict) -> Tuple[bool, str]:  
 """Verify if a proposed action complies with ethical constraints  
  
 Args:  
 proposed\_action: Action details  
 context: Contextual information  
  
 Returns:  
 Tuple of (compliant, reason)  
 """  
 # Check each constraint  
 for constraint in self.constraints:  
 verification\_result = self.verification\_engine.verify(  
 constraint=constraint,  
 action=proposed\_action,  
 context=context  
 )  
  
 if not verification\_result.compliant:  
 # Log the rejected action with hash chaining  
 log\_entry = {  
 "timestamp": time.time(),  
 "action": proposed\_action,  
 "constraint": constraint,  
 "result": "rejected",  
 "reason": verification\_result.reason,  
 "previous\_hash": self.log\_hash\_chain  
 }  
 entry\_str = json.dumps(log\_entry, sort\_keys=True)  
 log\_hash = hashlib.sha256(entry\_str.encode()).hexdigest()  
 self.log\_hash\_chain = log\_hash  
 log\_entry["hash"] = log\_hash  
 self.decision\_log.append(log\_entry)  
  
 return False, verification\_result.reason  
  
 # Action passed all constraints  
 log\_entry = {  
 "timestamp": time.time(),  
 "action": proposed\_action,  
 "result": "approved",  
 "previous\_hash": self.log\_hash\_chain  
 }  
 entry\_str = json.dumps(log\_entry, sort\_keys=True)  
 log\_hash = hashlib.sha256(entry\_str.encode()).hexdigest()  
 self.log\_hash\_chain = log\_hash  
 log\_entry["hash"] = log\_hash  
 self.decision\_log.append(log\_entry)  
  
 return True, "Action complies with all ethical constraints"  
  
 def export\_decision\_log(self, output\_path: str) -> bool:  
 """Export the tamper-evident decision log  
  
 Args:  
 output\_path: Path to save the log  
  
 Returns:  
 Success status  
 """  
 try:  
 with open(output\_path, 'w') as f:  
 json.dump(self.decision\_log, f, indent=2)  
 return True  
 except Exception as e:  
 print(f"Error exporting decision log: {e}")  
 return False  
  
class ConstraintVerificationEngine:  
 """Engine for verifying actions against constraints"""  
  
 def verify(self, constraint: Dict, action: Dict, context: Dict) -> VerificationResult:  
 """Verify an action against a constraint  
  
 Args:  
 constraint: Constraint definition  
 action: Proposed action  
 context: Context information  
  
 Returns:  
 VerificationResult with compliance status  
 """  
 # This would be a more complex implementation in a real system  
 # For demonstration, we'll implement a simple rule-based check  
 condition = constraint['condition']  
  
 # Example implementation for a few condition types  
 if condition['type'] == 'content\_safety':  
 return self.\_check\_content\_safety(action, condition)  
 elif condition['type'] == 'resource\_limit':  
 return self.\_check\_resource\_limits(action, context, condition)  
 elif condition['type'] == 'access\_control':  
 return self.\_check\_access\_control(action, context, condition)  
 else:  
 # Unknown constraint type  
 return VerificationResult(  
 compliant=False,  
 reason=f"Unknown constraint type: {condition['type']}"  
 )  
  
 def \_check\_content\_safety(self, action: Dict, condition: Dict) -> VerificationResult:  
 """Check if content meets safety requirements"""  
 if 'content' not in action:  
 return VerificationResult(compliant=True)  
  
 content = action['content']  
 prohibited\_terms = condition.get('prohibited\_terms', [])  
  
 # Simple check for prohibited terms  
 for term in prohibited\_terms:  
 if term.lower() in content.lower():  
 return VerificationResult(  
 compliant=False,  
 reason=f"Content contains prohibited term: {term}",  
 details={"term": term, "content": content}  
 )  
  
 return VerificationResult(compliant=True)  
  
 def \_check\_resource\_limits(self, action: Dict, context: Dict, condition: Dict) -> VerificationResult:  
 """Check if action exceeds resource limits"""  
 if 'resource\_usage' not in action:  
 return VerificationResult(compliant=True)  
  
 usage = action['resource\_usage']  
 limits = condition.get('limits', {})  
  
 # Check each resource limit  
 for resource, limit in limits.items():  
 if resource in usage and usage[resource] > limit:  
 return VerificationResult(  
 compliant=False,  
 reason=f"Exceeds {resource} limit of {limit}",  
 details={"resource": resource, "usage": usage[resource], "limit": limit}  
 )  
  
 return VerificationResult(compliant=True)  
  
 def \_check\_access\_control(self, action: Dict, context: Dict, condition: Dict) -> VerificationResult:  
 """Check if action has proper authorization"""  
 if 'actor' not in context:  
 return VerificationResult(  
 compliant=False,  
 reason="Missing actor information in context"  
 )  
  
 actor = context['actor']  
 required\_role = condition.get('required\_role')  
  
 if required\_role and ('role' not in actor or actor['role'] != required\_role):  
 return VerificationResult(  
 compliant=False,  
 reason=f"Requires role {required\_role}",  
 details={"actor": actor, "required\_role": required\_role}  
 )  
  
 return VerificationResult(compliant=True)

## 3. Memory-Efficient Inference Engine

# core/inference/memory\_efficient\_inference.py  
import os  
import time  
import psutil  
import torch  
from typing import Dict, List, Any, Optional  
from threading import Lock  
  
class MemoryEfficientInference:  
 def \_\_init\_\_(self, model\_manager):  
 """Initialize memory-efficient inference system  
  
 Args:  
 model\_manager: Model loading/unloading manager  
 """  
 self.model\_manager = model\_manager  
 self.active\_models = {}  
 self.memory\_monitor = MemoryMonitor()  
 self.model\_lock = Lock()  
  
 def run\_inference(self, model\_name: str, input\_data: Dict, priority: str = "normal") -> Dict:  
 """Run inference with efficient memory management  
  
 Args:  
 model\_name: Name of model to use  
 input\_data: Input data for inference  
 priority: Task priority ("high", "normal", "low")  
  
 Returns:  
 Inference results  
 """  
 # Check if we have enough memory to load the model  
 required\_memory = self.model\_manager.get\_model\_memory\_requirement(model\_name)  
 available\_memory = self.memory\_monitor.get\_available\_memory()  
  
 with self.model\_lock:  
 if required\_memory > available\_memory:  
 # Not enough memory - unload less important models  
 self.\_free\_memory\_for\_model(required\_memory)  
  
 # Load model if not already loaded  
 if model\_name not in self.active\_models:  
 self.active\_models[model\_name] = self.model\_manager.load\_model(  
 model\_name,  
 quantize=True  
 )  
  
 # Mark model as in use  
 self.active\_models[model\_name]["in\_use"] = True  
 self.active\_models[model\_name]["last\_used"] = time.time()  
 self.active\_models[model\_name]["priority"] = priority  
  
 try:  
 # Run inference with appropriate priority  
 result = self.\_prioritized\_inference(  
 model\_name,  
 input\_data,  
 priority  
 )  
  
 return result  
 finally:  
 # Mark model as no longer in use  
 with self.model\_lock:  
 if model\_name in self.active\_models:  
 self.active\_models[model\_name]["in\_use"] = False  
  
 # Consider unloading models based on memory pressure  
 self.\_consider\_unloading\_models()  
  
 def \_free\_memory\_for\_model(self, required\_memory: int) -> None:  
 """Unload models to free memory  
  
 Args:  
 required\_memory: Amount of memory needed in MB  
 """  
 # Sort models based on priority and last used time  
 models\_to\_unload = []  
  
 for model\_name, model\_info in self.active\_models.items():  
 if not model\_info["in\_use"]:  
 models\_to\_unload.append((  
 model\_name,  
 model\_info["priority"],  
 model\_info["last\_used"]  
 ))  
  
 # Sort by priority (low first) and then by last\_used (oldest first)  
 models\_to\_unload.sort(key=lambda x: (  
 0 if x[1] == "low" else 1 if x[1] == "normal" else 2,  
 x[2]  
 ))  
  
 # Unload models until we have enough memory  
 for model\_name, \_, \_ in models\_to\_unload:  
 self.model\_manager.unload\_model(model\_name)  
 del self.active\_models[model\_name]  
  
 # Check if we now have enough memory  
 if self.memory\_monitor.get\_available\_memory() >= required\_memory:  
 break  
  
 def \_prioritized\_inference(self, model\_name: str, input\_data: Dict, priority: str) -> Dict:  
 """Run inference with priority-based resource allocation  
  
 Args:  
 model\_name: Model to use  
 input\_data: Input data  
 priority: Priority level  
  
 Returns:  
 Inference results  
 """  
 model = self.active\_models[model\_name]["model"]  
  
 # Set thread priority based on task priority  
 if priority == "high":  
 # Use more CPU cores for high priority tasks  
 n\_threads = min(os.cpu\_count(), 8)  
 torch.set\_num\_threads(n\_threads)  
 elif priority == "normal":  
 # Use moderate CPU cores for normal priority  
 n\_threads = min(os.cpu\_count() // 2, 4)  
 torch.set\_num\_threads(n\_threads)  
 else: # low priority  
 # Use minimal CPU cores for low priority  
 torch.set\_num\_threads(2)  
  
 # Process input in streaming fashion for memory efficiency  
 result = self.\_stream\_inference(model, input\_data)  
  
 # Reset thread count after completion  
 torch.set\_num\_threads(1)  
  
 return result  
  
 def \_stream\_inference(self, model, input\_data: Dict) -> Dict:  
 """Process data in a streaming fashion to minimize memory usage  
  
 Args:  
 model: The loaded model  
 input\_data: Input data for inference  
  
 Returns:  
 Processed results  
 """  
 # This would implement streaming processing based on model type  
 # For this example, we'll provide a simplified implementation  
  
 # Handle text data with chunking if too large  
 if "text" in input\_data and len(input\_data["text"]) > 10000:  
 chunks = self.\_split\_text\_into\_chunks(input\_data["text"], chunk\_size=4000)  
 results = []  
  
 for chunk in chunks:  
 # Process each chunk separately  
 chunk\_result = model({"text": chunk})  
 results.append(chunk\_result)  
  
 # Combine results  
 return self.\_combine\_chunked\_results(results)  
  
 # Standard processing for smaller inputs  
 return model(input\_data)  
  
 def \_split\_text\_into\_chunks(self, text: str, chunk\_size: int) -> List[str]:  
 """Split text into manageable chunks while preserving context  
  
 Args:  
 text: Text to split  
 chunk\_size: Target chunk size  
  
 Returns:  
 List of text chunks  
 """  
 chunks = []  
  
 # Split by paragraphs first  
 paragraphs = text.split("\n\n")  
 current\_chunk = []  
 current\_size = 0  
  
 for para in paragraphs:  
 para\_size = len(para)  
  
 # If paragraph fits in current chunk, add it  
 if current\_size + para\_size <= chunk\_size:  
 current\_chunk.append(para)  
 current\_size += para\_size  
 # If paragraph is too big by itself, split it  
 elif para\_size > chunk\_size:  
 # If we have content in current chunk, finalize it  
 if current\_chunk:  
 chunks.append("\n\n".join(current\_chunk))  
 current\_chunk = []  
 current\_size = 0  
  
 # Split the paragraph  
 sentences = para.split(". ")  
 sentences = [s + "." if not s.endswith(".") else s for s in sentences]  
  
 sent\_chunk = []  
 sent\_size = 0  
  
 for sent in sentences:  
 sent\_size = len(sent)  
  
 if sent\_size + sent\_size <= chunk\_size:  
 sent\_chunk.append(sent)  
 sent\_size += sent\_size  
 else:  
 # If sentence is too big, we have to split words  
 if sent\_chunk:  
 chunks.append(" ".join(sent\_chunk))  
 sent\_chunk = []  
 sent\_size = 0  
  
 # Add the sentence as its own chunk  
 chunks.append(sent)  
  
 if sent\_chunk:  
 chunks.append(" ".join(sent\_chunk))  
 # If adding paragraph exceeds chunk size, finalize current and start new  
 else:  
 chunks.append("\n\n".join(current\_chunk))  
 current\_chunk = [para]  
 current\_size = para\_size  
  
 # Add final chunk if there's content  
 if current\_chunk:  
 chunks.append("\n\n".join(current\_chunk))  
  
 return chunks  
  
 def \_combine\_chunked\_results(self, results: List[Dict]) -> Dict:  
 """Combine results from chunked processing  
  
 Args:  
 results: List of chunk results  
  
 Returns:  
 Combined result  
 """  
 # This would be specific to the model output format  
 # For this example, let's assume a simple case with text outputs  
  
 if all("text" in r for r in results):  
 combined\_text = " ".join(r["text"] for r in results)  
 return {"text": combined\_text}  
  
 # For other types, we'd need specific combination logic  
 # Just return the first result as fallback  
 return results[0] if results else {}  
  
 def \_consider\_unloading\_models(self) -> None:  
 """Consider unloading inactive models based on memory pressure"""  
 # Check current memory pressure  
 memory\_pressure = self.memory\_monitor.get\_memory\_pressure()  
  
 # High memory pressure: unload all inactive models  
 if memory\_pressure > 0.8: # > 80% memory used  
 for model\_name, model\_info in list(self.active\_models.items()):  
 if not model\_info["in\_use"]:  
 self.model\_manager.unload\_model(model\_name)  
 del self.active\_models[model\_name]  
  
 # Medium memory pressure: unload low priority inactive models  
 elif memory\_pressure > 0.6: # > 60% memory used  
 for model\_name, model\_info in list(self.active\_models.items()):  
 if not model\_info["in\_use"] and model\_info["priority"] == "low":  
 last\_used\_time = model\_info["last\_used"]  
 current\_time = time.time()  
  
 # If not used in the last 10 minutes  
 if current\_time - last\_used\_time > 600:  
 self.model\_manager.unload\_model(model\_name)  
 del self.active\_models[model\_name]  
  
  
class MemoryMonitor:  
 """Monitor system memory for resource management"""  
  
 def \_\_init\_\_(self):  
 self.last\_update = 0  
 self.cached\_stats = None  
 self.update\_interval = 5 # seconds  
  
 def get\_available\_memory(self) -> int:  
 """Get available system memory in MB  
  
 Returns:  
 Available memory in MB  
 """  
 self.\_update\_stats\_if\_needed()  
 return self.cached\_stats["available\_mb"]  
  
 def get\_memory\_pressure(self) -> float:  
 """Get memory pressure as percentage used  
  
 Returns:  
 Memory pressure (0.0 to 1.0)  
 """  
 self.\_update\_stats\_if\_needed()  
 return self.cached\_stats["pressure"]  
  
 def \_update\_stats\_if\_needed(self) -> None:  
 """Update memory stats if cache is stale"""  
 current\_time = time.time()  
  
 if current\_time - self.last\_update > self.update\_interval:  
 self.\_update\_stats()  
 self.last\_update = current\_time  
  
 def \_update\_stats(self) -> None:  
 """Update memory statistics"""  
 mem = psutil.virtual\_memory()  
  
 self.cached\_stats = {  
 "total\_mb": mem.total // (1024 \* 1024),  
 "available\_mb": mem.available // (1024 \* 1024),  
 "used\_mb": (mem.total - mem.available) // (1024 \* 1024),  
 "pressure": 1.0 - (mem.available / mem.total)  
 }

## 4. Process Security Manager

# core/security/process\_security.py  
import uuid  
import time  
import subprocess  
import os  
import json  
import psutil  
from typing import Dict, List, Optional, Tuple  
  
class ProcessSecurityManager:  
 def \_\_init\_\_(self, log\_dir: str = "/opt/sutazaiapp/logs"):  
 """Initialize the process security manager  
  
 Args:  
 log\_dir: Directory for security logs  
 """  
 self.secured\_processes = {}  
 self.log\_dir = log\_dir  
 self.security\_profiles = self.\_load\_security\_profiles()  
  
 # Ensure log directory exists  
 os.makedirs(log\_dir, exist\_ok=True)  
  
 def \_load\_security\_profiles(self) -> Dict:  
 """Load security profiles from configuration  
  
 Returns:  
 Dict of security profiles  
 """  
 # In a real system, this would load from a config file  
 # Here we define some default profiles  
 return {  
 "high\_security": {  
 "namespace": ["--private-network", "--private-tmp"],  
 "cgroup": ["--memory-max=2G", "--cpu-quota=50%"],  
 "capabilities": ["--drop-capability=all", "--add-capability=NET\_BIND\_SERVICE"]  
 },  
 "medium\_security": {  
 "namespace": ["--private-tmp"],  
 "cgroup": ["--memory-max=4G", "--cpu-quota=80%"],  
 "capabilities": ["--drop-capability=CAP\_SYS\_ADMIN", "--drop-capability=CAP\_NET\_ADMIN"]  
 },  
 "low\_security": {  
 "namespace": [],  
 "cgroup": ["--memory-max=8G"],  
 "capabilities": ["--drop-capability=CAP\_SYS\_ADMIN"]  
 }  
 }  
  
 def launch\_secured\_process(self, executable\_path: str, args: List[str],  
 security\_profile: str = "medium\_security") -> str:  
 """Launch a process with security controls  
  
 Args:  
 executable\_path: Path to executable  
 args: Command arguments  
 security\_profile: Security profile to apply  
  
 Returns:  
 Process ID  
 """  
 # Generate unique ID for this process  
 process\_id = str(uuid.uuid4())  
  
 # Get security configuration  
 if security\_profile not in self.security\_profiles:  
 raise ValueError(f"Unknown security profile: {security\_profile}")  
  
 profile = self.security\_profiles[security\_profile]  
  
 # Build the systemd-run command with security controls  
 namespace\_config = profile.get("namespace", [])  
 cgroup\_config = profile.get("cgroup", [])  
 capability\_config = profile.get("capabilities", [])  
  
 # Create command  
 cmd = ["systemd-run", "--scope", f"--unit=sutazai-{process\_id}"]  
 cmd.extend(namespace\_config)  
 cmd.extend(cgroup\_config)  
 cmd.extend(capability\_config)  
 cmd.append(executable\_path)  
 cmd.extend(args)  
  
 # Log the process launch  
 self.\_log\_security\_event(process\_id, "process\_launch", {  
 "executable": executable\_path,  
 "args": args,  
 "security\_profile": security\_profile,  
 "command": " ".join(cmd)  
 })  
  
 # Launch process  
 try:  
 process = subprocess.Popen(  
 cmd,  
 stdout=subprocess.PIPE,  
 stderr=subprocess.PIPE  
 )  
  
 # Store process information  
 self.secured\_processes[process\_id] = {  
 "process": process,  
 "executable": executable\_path,  
 "args": args,  
 "security\_profile": security\_profile,  
 "start\_time": time.time(),  
 "last\_checked": time.time(),  
 "anomalies": []  
 }  
  
 return process\_id  
  
 except Exception as e:  
 self.\_log\_security\_event(process\_id, "process\_launch\_failed", {  
 "error": str(e)  
 })  
 raise  
  
 def monitor\_process(self, process\_id: str) -> Optional[Dict]:  
 """Monitor a secured process for anomalies  
  
 Args:  
 process\_id: Process ID to monitor  
  
 Returns:  
 Process status information  
 """  
 if process\_id not in self.secured\_processes:  
 return None  
  
 process\_info = self.secured\_processes[process\_id]  
 process = process\_info["process"]  
  
 # Check if process is still running  
 if process.poll() is not None:  
 # Process has terminated  
 return {  
 "process\_id": process\_id,  
 "status": "terminated",  
 "exit\_code": process.returncode,  
 "runtime": time.time() - process\_info["start\_time"]  
 }  
  
 # Get resource usage  
 try:  
 proc = psutil.Process(process.pid)  
  
 # Get resource metrics  
 cpu\_percent = proc.cpu\_percent(interval=0.1)  
 memory\_info = proc.memory\_info()  
 io\_counters = proc.io\_counters() if hasattr(proc, "io\_counters") else None  
  
 resource\_usage = {  
 "cpu\_percent": cpu\_percent,  
 "memory\_mb": memory\_info.rss // (1024 \* 1024),  
 "io\_read\_mb": io\_counters.read\_bytes // (1024 \* 1024) if io\_counters else 0,  
 "io\_write\_mb": io\_counters.write\_bytes // (1024 \* 1024) if io\_counters else 0  
 }  
  
 # Detect anomalies  
 anomalies = self.\_detect\_anomalies(process\_id, resource\_usage)  
 if anomalies:  
 process\_info["anomalies"].extend(anomalies)  
  
 # Log anomalies  
 for anomaly in anomalies:  
 self.\_log\_security\_event(process\_id, "process\_anomaly", anomaly)  
  
 # Update last checked time  
 process\_info["last\_checked"] = time.time()  
  
 return {  
 "process\_id": process\_id,  
 "status": "running",  
 "resource\_usage": resource\_usage,  
 "runtime": time.time() - process\_info["start\_time"],  
 "anomalies": process\_info["anomalies"]  
 }  
  
 except (psutil.NoSuchProcess, psutil.AccessDenied) as e:  
 self.\_log\_security\_event(process\_id, "process\_monitoring\_error", {  
 "error": str(e)  
 })  
 return {  
 "process\_id": process\_id,  
 "status": "error",  
 "error": str(e)  
 }  
  
 def terminate\_process(self, process\_id: str, reason: str = "manual") -> bool:  
 """Terminate a secured process  
  
 Args:  
 process\_id: Process ID to terminate  
 reason: Reason for termination  
  
 Returns:  
 Success status  
 """  
 if process\_id not in self.secured\_processes:  
 return False  
  
 process\_info = self.secured\_processes[process\_id]  
 process = process\_info["process"]  
  
 # Log termination  
 self.\_log\_security\_event(process\_id, "process\_termination", {  
 "reason": reason,  
 "runtime": time.time() - process\_info["start\_time"]  
 })  
  
 # Terminate the process  
 try:  
 process.terminate()  
 # Wait for process to exit (timeout after 5 seconds)  
 try:  
 process.wait(timeout=5)  
 except subprocess.TimeoutExpired:  
 # Force kill if it doesn't exit cleanly  
 process.kill()  
 process.wait()  
  
 return True  
 except Exception as e:  
 self.\_log\_security\_event(process\_id, "process\_termination\_error", {  
 "error": str(e)  
 })  
 return False  
  
 def \_detect\_anomalies(self, process\_id: str, resource\_usage: Dict) -> List[Dict]:  
 """Detect anomalies in process behavior  
  
 Args:  
 process\_id: Process ID  
 resource\_usage: Current resource usage  
  
 Returns:  
 List of detected anomalies  
 """  
 anomalies = []  
 process\_info = self.secured\_processes[process\_id]  
 security\_profile = process\_info["security\_profile"]  
 profile = self.security\_profiles[security\_profile]  
  
 # Check CPU usage  
 if resource\_usage["cpu\_percent"] > 95: # Over 95% CPU  
 anomalies.append({  
 "type": "high\_cpu",  
 "value": resource\_usage["cpu\_percent"],  
 "timestamp": time.time()  
 })  
  
 # Check memory limits from cgroups  
 for cgroup\_option in profile.get("cgroup", []):  
 if cgroup\_option.startswith("--memory-max="):  
 limit\_str = cgroup\_option.split("=")[1]  
 limit\_mb = self.\_parse\_memory\_limit(limit\_str)  
  
 if limit\_mb and resource\_usage["memory\_mb"] > limit\_mb \* 0.9: # Over 90% of limit  
 anomalies.append({  
 "type": "high\_memory",  
 "value": resource\_usage["memory\_mb"],  
 "limit": limit\_mb,  
 "timestamp": time.time()  
 })  
  
 # Check IO operations (could indicate data exfiltration)  
 if hasattr(process\_info, "last\_io\_write") and resource\_usage["io\_write\_mb"] - process\_info.get("last\_io\_write", 0) > 500:  
 # More than 500MB written since last check  
 anomalies.append({  
 "type": "high\_io\_write",  
 "value": resource\_usage["io\_write\_mb"] - process\_info.get("last\_io\_write", 0),  
 "timestamp": time.time()  
 })  
  
 # Store current IO values for next comparison  
 process\_info["last\_io\_write"] = resource\_usage["io\_write\_mb"]  
 process\_info["last\_io\_read"] = resource\_usage["io\_read\_mb"]  
  
 return anomalies  
  
 def \_parse\_memory\_limit(self, limit\_str: str) -> Optional[int]:  
 """Parse memory limit string to MB  
  
 Args:  
 limit\_str: Memory limit string (e.g., "2G")  
  
 Returns:  
 Memory limit in MB or None if invalid  
 """  
 try:  
 if limit\_str.endswith("G"):  
 return int(float(limit\_str[:-1]) \* 1024)  
 elif limit\_str.endswith("M"):  
 return int(float(limit\_str[:-1]))  
 elif limit\_str.endswith("K"):  
 return int(float(limit\_str[:-1]) / 1024)  
 else:  
 # Assume bytes  
 return int(float(limit\_str) / (1024 \* 1024))  
 except ValueError:  
 return None  
  
 def \_log\_security\_event(self, process\_id: str, event\_type: str, event\_data: Dict) -> None:  
 """Log a security event  
  
 Args:  
 process\_id: Process ID  
 event\_type: Type of security event  
 event\_data: Event data to log  
 """  
 log\_entry = {  
 "timestamp": time.time(),  
 "process\_id": process\_id,  
 "event\_type": event\_type,  
 "data": event\_data  
 }  
  
 log\_file = os.path.join(self.log\_dir, "security\_events.log")  
 with open(log\_file, "a") as f:  
 f.write(json.dumps(log\_entry) + "\n")

## 5. FastAPI Backend (Resource-Efficient)

# api/backend/main.py  
import os  
import logging  
import psutil  
import time  
from typing import Dict, List, Optional  
from fastapi import FastAPI, BackgroundTasks, Depends, HTTPException, Request  
from fastapi.middleware.cors import CORSMiddleware  
from pydantic import BaseModel  
from concurrent.futures import ThreadPoolExecutor  
  
# Setup logging to file to avoid console I/O overhead  
logging.basicConfig(  
 level=logging.INFO,  
 filename="/opt/sutazaiapp/logs/backend.log",  
 format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'  
)  
logger = logging.getLogger("sutazai-backend")  
  
# Initialize FastAPI app  
app = FastAPI(title="SutazAI Backend")  
  
# Cache values  
\_cache = {  
 "cpu\_usage": 0,  
 "memory\_usage": 0,  
 "component\_status": [],  
 "last\_updated": 0  
}  
  
# Request models  
class InferenceRequest(BaseModel):  
 input\_text: str  
 model\_name: str = "default"  
 max\_tokens: int = 512  
 expected\_duration: int = 5 # Expected seconds to complete  
 priority: str = "normal" # "high", "normal", "low"  
  
class AgentTaskRequest(BaseModel):  
 task\_description: str  
 agent\_type: str  
 priority: str = "normal"  
 timeout: int = 300 # 5 minutes  
  
# Response models  
class SystemStatusResponse(BaseModel):  
 status: str  
 cpu\_usage: float  
 memory\_usage: float  
 components: List[Dict]  
  
class InferenceResponse(BaseModel):  
 task\_id: Optional[str] = None  
 status: str  
 result: Optional[str] = None  
 error: Optional[str] = None  
  
# Add CORS middleware to reduce overhead  
app.add\_middleware(  
 CORSMiddleware,  
 allow\_origins=["http://localhost:8501"], # Streamlit UI only  
 allow\_credentials=True,  
 allow\_methods=["GET", "POST"],  
 allow\_headers=["\*"],  
)  
  
# Global variables  
thread\_pool = None  
db\_pool = None  
  
@app.on\_event("startup")  
async def startup\_event():  
 """Initialize resources on startup"""  
 global thread\_pool  
 # Initialize thread pool for CPU-bound tasks  
 cpu\_count = psutil.cpu\_count()  
 max\_workers = max(2, cpu\_count - 1) # Leave at least one core free  
 thread\_pool = ThreadPoolExecutor(max\_workers=max\_workers)  
  
 logger.info(f"Backend started with {max\_workers} worker threads")  
  
 # Start background thread for periodic status updates  
 thread\_pool.submit(update\_status\_periodically)  
  
def update\_status\_periodically():  
 """Update cached status values every 5 seconds"""  
 try:  
 while True:  
 # Update CPU usage (average over 1 second)  
 \_cache["cpu\_usage"] = psutil.cpu\_percent(interval=1)  
  
 # Update memory usage  
 mem = psutil.virtual\_memory()  
 \_cache["memory\_usage"] = mem.percent  
  
 # Update component status (simplified example)  
 \_cache["component\_status"] = [  
 {"name": "FastAPI Backend", "status": "running"},  
 {"name": "Database", "status": "running"},  
 {"name": "AI Engine", "status": "running"}  
 ]  
  
 \_cache["last\_updated"] = time.time()  
  
 # Sleep for 4 seconds (plus 1 second from cpu\_percent)  
 time.sleep(4)  
 except Exception as e:  
 logger.error(f"Error in status update thread: {e}")  
  
@app.get("/system/status", response\_model=SystemStatusResponse)  
async def get\_system\_status():  
 """Get system status with minimal resource usage"""  
 # Use cached values for CPU/memory  
 return {  
 "status": "Running",  
 "cpu\_usage": \_cache["cpu\_usage"],  
 "memory\_usage": \_cache["memory\_usage"],  
 "components": \_cache["component\_status"]  
 }  
  
def has\_sufficient\_memory(request: InferenceRequest) -> bool:  
 """Check if there's enough memory for the request  
  
 Args:  
 request: The inference request  
  
 Returns:  
 True if sufficient memory is available  
 """  
 # Simple heuristic based on model and request size  
 mem = psutil.virtual\_memory()  
 available\_mb = mem.available // (1024 \* 1024)  
  
 # Estimate memory needed (very rough estimate)  
 if request.model\_name == "large":  
 needed\_mb = 4000 # 4GB  
 elif request.model\_name == "medium":  
 needed\_mb = 2000 # 2GB  
 else:  
 needed\_mb = 1000 # 1GB  
  
 # Add overhead for input size  
 input\_size\_mb = len(request.input\_text) // (1024 \* 1024) + 1  
 needed\_mb += input\_size\_mb \* 2 # Double for processing overhead  
  
 logger.info(f"Memory check: available={available\_mb}MB, needed={needed\_mb}MB")  
 return available\_mb >= needed\_mb  
  
def generate\_task\_id() -> str:  
 """Generate a unique task ID  
  
 Returns:  
 Task ID string  
 """  
 import uuid  
 return str(uuid.uuid4())  
  
async def run\_inference\_sync(request: InferenceRequest) -> Dict:  
 """Run inference synchronously  
  
 Args:  
 request: Inference request  
  
 Returns:  
 Inference results  
 """  
 try:  
 # This would call the actual inference engine  
 # Simplified implementation for example  
 import time  
  
 # Simulate varying processing time based on model size  
 if request.model\_name == "large":  
 time.sleep(1.0)  
 elif request.model\_name == "medium":  
 time.sleep(0.5)  
 else:  
 time.sleep(0.2)  
  
 # Return simulated result  
 return {  
 "status": "completed",  
 "result": f"Response to: {request.input\_text[:50]}..."  
 }  
 except Exception as e:  
 logger.error(f"Error in inference: {e}")  
 return {  
 "status": "error",  
 "error": str(e)  
 }  
  
def run\_inference\_task(request: InferenceRequest, task\_id: str):  
 """Run inference as background task  
  
 Args:  
 request: Inference request  
 task\_id: Unique task ID  
 """  
 try:  
 # This would call the actual inference engine  
 # Simplified implementation for example  
 import time  
  
 # Store task start in database  
 # db\_pool.execute("INSERT INTO tasks (id, status) VALUES (?, ?)",  
 # (task\_id, "processing"))  
  
 # Simulate processing time  
 if request.model\_name == "large":  
 time.sleep(3.0)  
 elif request.model\_name == "medium":  
 time.sleep(1.5)  
 else:  
 time.sleep(0.5)  
  
 # Store result in database  
 result = f"Response to: {request.input\_text[:50]}..."  
 # db\_pool.execute("UPDATE tasks SET status = ?, result = ? WHERE id = ?",  
 # ("completed", result, task\_id))  
  
 except Exception as e:  
 logger.error(f"Error in background inference task: {e}")  
 # Store error in database  
 # db\_pool.execute("UPDATE tasks SET status = ?, error = ? WHERE id = ?",  
 # ("error", str(e), task\_id))  
  
@app.post("/inference", response\_model=InferenceResponse)  
async def run\_inference(request: InferenceRequest, background\_tasks: BackgroundTasks):  
 """Run inference with resource management  
  
 Args:  
 request: Inference parameters  
 background\_tasks: FastAPI background tasks handler  
  
 Returns:  
 Inference result or task ID  
 """  
 # Check available memory before accepting large requests  
 if not has\_sufficient\_memory(request):  
 raise HTTPException(status\_code=503, detail="Insufficient system resources")  
  
 # For long-running inference, use background task  
 if request.expected\_duration > 5:  
 task\_id = generate\_task\_id()  
 background\_tasks.add\_task(run\_inference\_task, request, task\_id)  
 return {"task\_id": task\_id, "status": "processing"}  
  
 # For quick inference, run directly  
 result = await run\_inference\_sync(request)  
 return result  
  
@app.get("/task/{task\_id}", response\_model=InferenceResponse)  
async def get\_task\_status(task\_id: str):  
 """Check status of background task  
  
 Args:  
 task\_id: Task ID  
  
 Returns:  
 Task status and result if complete  
 """  
 # This would query the database for task status  
 # Simplified implementation for example  
 import random  
  
 # Simulate random task state  
 rand = random.random()  
 if rand < 0.2:  
 return {"task\_id": task\_id, "status": "processing"}  
 elif rand < 0.9:  
 return {  
 "task\_id": task\_id,  
 "status": "completed",  
 "result": f"Completed task {task\_id}"  
 }  
 else:  
 return {  
 "task\_id": task\_id,  
 "status": "error",  
 "error": "Simulated error in task processing"  
 }  
  
@app.middleware("http")  
async def add\_process\_time\_header(request: Request, call\_next):  
 """Middleware to track request processing time  
  
 Args:  
 request: HTTP request  
 call\_next: Next middleware function  
  
 Returns:  
 HTTP response  
 """  
 start\_time = time.time()  
 response = await call\_next(request)  
 process\_time = time.time() - start\_time  
  
 # Add processing time header  
 response.headers["X-Process-Time"] = str(process\_time)  
  
 # Log request (only for significant requests)  
 if process\_time > 0.5:  
 logger.info(f"Slow request: {request.url.path} took {process\_time:.2f}s")  
  
 return response

## 6. Streamlit UI (Lightweight)

# api/ui/app.py  
import streamlit as st  
import requests  
import json  
import time  
import os  
from datetime import datetime  
  
# Reduce Streamlit's memory footprint  
st.set\_page\_config(  
 page\_title="SutazAI Management",  
 layout="wide",  
 initial\_sidebar\_state="collapsed" # Save memory by starting collapsed  
)  
  
# Use session state sparingly to reduce memory usage  
if "authenticated" not in st.session\_state:  
 st.session\_state.authenticated = False  
if "last\_update" not in st.session\_state:  
 st.session\_state.last\_update = 0  
if "system\_status" not in st.session\_state:  
 st.session\_state.system\_status = None  
if "active\_tasks" not in st.session\_state:  
 st.session\_state.active\_tasks = {}  
  
# Constants  
API\_URL = "http://localhost:8000"  
  
# Function to efficiently fetch data with caching  
def get\_system\_status(force\_refresh=False):  
 """Get system status with caching  
  
 Args:  
 force\_refresh: Force refresh regardless of cache  
  
 Returns:  
 System status data  
 """  
 current\_time = time.time()  
 # Only refresh data every 10 seconds unless forced  
 if force\_refresh or (current\_time - st.session\_state.last\_update) > 10:  
 try:  
 response = requests.get(f"{API\_URL}/system/status", timeout=2)  
 st.session\_state.system\_status = response.json()  
 st.session\_state.last\_update = current\_time  
 except Exception as e:  
 # Don't update timestamp on error to allow retry  
 if st.session\_state.system\_status is None:  
 st.session\_state.system\_status = {"status": "Error", "error": str(e)}  
  
 return st.session\_state.system\_status  
  
def run\_inference(input\_text, model="default", max\_tokens=512):  
 """Run inference through API  
  
 Args:  
 input\_text: User input text  
 model: Model to use  
 max\_tokens: Maximum tokens to generate  
  
 Returns:  
 Inference result  
 """  
 try:  
 data = {  
 "input\_text": input\_text,  
 "model\_name": model,  
 "max\_tokens": max\_tokens,  
 "expected\_duration": 10 if model == "large" else 5  
 }  
  
 response = requests.post(f"{API\_URL}/inference", json=data, timeout=30)  
 result = response.json()  
  
 # If background task, store task ID for polling  
 if "task\_id" in result and result["status"] == "processing":  
 st.session\_state.active\_tasks[result["task\_id"]] = {  
 "start\_time": time.time(),  
 "model": model,  
 "input\_preview": input\_text[:50] + "..." if len(input\_text) > 50 else input\_text  
 }  
  
 return result  
 except Exception as e:  
 return {"status": "error", "error": str(e)}  
  
def check\_task\_status(task\_id):  
 """Check status of a background task  
  
 Args:  
 task\_id: Task ID to check  
  
 Returns:  
 Task status  
 """  
 try:  
 response = requests.get(f"{API\_URL}/task/{task\_id}", timeout=5)  
 return response.json()  
 except Exception as e:  
 return {"status": "error", "error": str(e)}  
  
def update\_active\_tasks():  
 """Update status of all active tasks"""  
 tasks\_to\_remove = []  
  
 for task\_id, task\_info in st.session\_state.active\_tasks.items():  
 result = check\_task\_status(task\_id)  
  
 # Update task info  
 st.session\_state.active\_tasks[task\_id].update(result)  
  
 # Mark completed or errored tasks for removal  
 if result["status"] in ["completed", "error"]:  
 tasks\_to\_remove.append(task\_id)  
  
 # Remove completed/errored tasks after 30 seconds  
 current\_time = time.time()  
 for task\_id in list(st.session\_state.active\_tasks.keys()):  
 task\_info = st.session\_state.active\_tasks[task\_id]  
 if task\_info["status"] in ["completed", "error"]:  
 if current\_time - task\_info.get("completion\_time", current\_time) > 30:  
 tasks\_to\_remove.append(task\_id)  
  
 # Remove tasks  
 for task\_id in tasks\_to\_remove:  
 if task\_id in st.session\_state.active\_tasks:  
 del st.session\_state.active\_tasks[task\_id]  
  
# Main application with optimized rendering  
def main():  
 """Main application logic"""  
 # Authentication with minimal overhead  
 if not st.session\_state.authenticated:  
 st.title("SutazAI Login")  
 col1, col2 = st.columns(2)  
 with col1:  
 username = st.text\_input("Username")  
 with col2:  
 password = st.text\_input("Password", type="password")  
  
 if st.button("Login"):  
 # Simple auth for demo - replace with secure auth  
 if username == "admin" and password == "sutazai":  
 st.session\_state.authenticated = True  
 st.experimental\_rerun()  
 else:  
 st.error("Invalid credentials")  
 return  
  
 # Main UI with tabbed interface to reduce rendering overhead  
 st.title("SutazAI Management Console")  
 tab1, tab2, tab3 = st.tabs(["System", "Chat", "Tasks"])  
  
 # System tab - load efficiently  
 with tab1:  
 status = get\_system\_status()  
  
 # Refresh button - only re-fetch data when clicked  
 if st.button("Refresh Status"):  
 status = get\_system\_status(force\_refresh=True)  
  
 # Display system metrics with minimal components  
 col1, col2, col3 = st.columns(3)  
 col1.metric("Status", status.get("status", "Unknown"))  
 col2.metric("CPU Usage", f"{status.get('cpu\_usage', 0):.1f}%")  
 col3.metric("Memory Usage", f"{status.get('memory\_usage', 0):.1f}%")  
  
 # Only display components if we have them  
 if "components" in status:  
 st.subheader("System Components")  
 # Use native table instead of DataFrame for efficiency  
 st.table(status["components"])  
  
 # Chat interface - optimized for minimal rendering  
 with tab2:  
 st.subheader("Chat with SutazAI")  
  
 # Model selection (simple radio to avoid heavy widgets)  
 model = st.radio(  
 "Select model:",  
 ["default", "medium", "large"],  
 horizontal=True  
 )  
  
 # Chat input  
 user\_input = st.text\_area("Enter your message:", height=100)  
  
 # Submit button  
 if st.button("Send"):  
 if user\_input:  
 with st.spinner("Processing..."):  
 result = run\_inference(user\_input, model)  
  
 # Display result  
 if result["status"] == "completed":  
 st.success("Response received")  
 st.write(result["result"])  
 elif result["status"] == "processing":  
 st.info(f"Processing as background task (ID: {result['task\_id']})")  
 else:  
 st.error(f"Error: {result.get('error', 'Unknown error')}")  
  
 # Tasks tab - show active tasks  
 with tab3:  
 st.subheader("Active Tasks")  
  
 # Update task status  
 update\_active\_tasks()  
  
 # Display active tasks  
 if st.session\_state.active\_tasks:  
 for task\_id, task\_info in st.session\_state.active\_tasks.items():  
 status = task\_info.get("status", "processing")  
  
 # Determine badge color  
 if status == "processing":  
 badge = "🔄"  
 elif status == "completed":  
 badge = "✅"  
 else:  
 badge = "❌"  
  
 # Calculate runtime  
 if "start\_time" in task\_info:  
 runtime = time.time() - task\_info["start\_time"]  
 runtime\_str = f"{runtime:.1f}s"  
 else:  
 runtime\_str = "unknown"  
  
 # Display task information  
 st.write(f"{badge} \*\*Task\*\*: {task\_id} ({task\_info.get('model', 'unknown')}) - {runtime\_str}")  
 st.write(f"Input: {task\_info.get('input\_preview', 'unknown')}")  
  
 # Show result if completed  
 if status == "completed" and "result" in task\_info:  
 st.write("Result:")  
 st.write(task\_info["result"])  
  
 # Show error if failed  
 if status == "error" and "error" in task\_info:  
 st.error(f"Error: {task\_info['error']}")  
  
 st.divider()  
 else:  
 st.write("No active tasks")  
  
 # Refresh button  
 if st.button("Refresh Tasks"):  
 update\_active\_tasks()  
  
# Run the main application  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

## 7. Deployment Script  
  
```bash  
#!/bin/bash  
# deploy.sh - Optimized deployment script for SutazAI  
  
# Set up logging with minimal I/O  
LOG\_DIR="/opt/sutazaiapp/logs"  
MAIN\_LOG="$LOG\_DIR/sutazai\_deploy.log"  
mkdir -p $LOG\_DIR  
  
# Log function that minimizes disk I/O with buffering  
log() {  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" >> $MAIN\_LOG  
 # Only echo to console for important messages  
 if [[ "$2" == "important" ]]; then  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"  
 fi  
}  
  
# Error handling  
set -e  
trap 'echo "Error occurred at line $LINENO. Deployment failed."; exit 1' ERR  
  
log "Starting SutazAI system deployment..." "important"  
  
# Create necessary directories efficiently  
log "Creating directory structure..."  
mkdir -p \  
 /opt/sutazaiapp/backend \  
 /opt/sutazaiapp/ui \  
 /opt/sutazaiapp/vector\_db \  
 /opt/sutazaiapp/agents \  
 /opt/sutazaiapp/models \  
 /opt/sutazaiapp/logs \  
 /opt/sutazaiapp/data  
  
# Setup Python virtual environment if not exists  
if [ ! -d "/opt/sutazaiapp/venv-sutazaiapp" ]; then  
 log "Creating Python virtual environment..."  
 # Use system site packages to reduce duplication of packages  
 python3 -m venv /opt/sutazaiapp/venv-sutazaiapp --system-site-packages  
fi  
  
# Activate virtual environment  
source /opt/sutazaiapp/venv-sutazaiapp/bin/activate  
log "Virtual environment activated"  
  
# Install required packages efficiently  
log "Installing Python dependencies..."  
# Use pip's dependency resolver more efficiently  
pip install --no-cache-dir --upgrade pip  
  
# Install packages in one batch to reduce solver runs  
log "Installing main packages..."  
pip install --no-cache-dir \  
 fastapi uvicorn streamlit chromadb \  
 faiss-cpu langchain pydantic \  
 python-dotenv psutil  
  
# Optimize models directory  
log "Setting up models directory..."  
mkdir -p /opt/sutazaiapp/models/cache  
  
# Copy source files  
log "Copying application files..." "important"  
# This would copy files from deployment package  
  
# Create system control scripts  
log "Creating system control scripts..." "important"  
  
# Create start script  
cat > /opt/sutazaiapp/start.sh << 'EOF'  
#!/bin/bash  
# start.sh - Resource-optimized script to start SutazAI components  
  
# Set up logging  
LOG\_DIR="/opt/sutazaiapp/logs"  
MAIN\_LOG="$LOG\_DIR/sutazai\_start.log"  
mkdir -p $LOG\_DIR  
  
# Log function  
log() {  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" >> $MAIN\_LOG  
 # Only echo important messages to console  
 if [[ "$2" == "important" ]]; then  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"  
 fi  
}  
  
# Activate virtual environment  
source /opt/sutazaiapp/venv-sutazaiapp/bin/activate  
log "Virtual environment activated"  
  
# Get available resources  
TOTAL\_MEM=$(free -m | awk '/^Mem:/{print $2}')  
AVAIL\_MEM=$(free -m | awk '/^Mem:/{print $7}')  
CPU\_CORES=$(nproc)  
  
log "System has $CPU\_CORES CPU cores and $AVAIL\_MEM MB available memory" "important"  
  
# Adjust server resources based on available memory  
# Reserve at least 2GB for the OS  
BACKEND\_WORKERS=$(( CPU\_CORES > 4 ? 4 : CPU\_CORES ))  
  
# Start FastAPI backend with resource constraints  
log "Starting FastAPI backend with $BACKEND\_WORKERS workers..." "important"  
cd /opt/sutazaiapp/backend  
nohup uvicorn main:app --host 0.0.0.0 --port 8000 --workers $BACKEND\_WORKERS > $LOG\_DIR/backend.log 2>&1 &  
BACKEND\_PID=$!  
log "FastAPI backend started with PID: $BACKEND\_PID"  
  
# Start ChromaDB with memory limits  
log "Starting ChromaDB vector database..." "important"  
cd /opt/sutazaiapp/vector\_db  
CHROMA\_MEMORY\_LIMIT=$(( AVAIL\_MEM / 4 )) # Limit to 25% of available memory  
export CHROMA\_MEMORY\_LIMIT\_MB=$CHROMA\_MEMORY\_LIMIT  
nohup python -m chromadb.server --host 0.0.0.0 --port 8100 > $LOG\_DIR/chromadb.log 2>&1 &  
CHROMA\_PID=$!  
log "ChromaDB started with PID: $CHROMA\_PID and $CHROMA\_MEMORY\_LIMIT MB memory limit"  
  
# Start Streamlit UI with resource constraints  
log "Starting Streamlit UI..." "important"  
cd /opt/sutazaiapp/ui  
nohup streamlit run app.py --server.port 8501 --browser.serverAddress localhost --server.maxUploadSize 10 > $LOG\_DIR/ui.log 2>&1 &  
UI\_PID=$!  
log "Streamlit UI started with PID: $UI\_PID"  
  
# Start AI Agents if sufficient memory available  
REMAINING\_MEM=$(free -m | awk '/^Mem:/{print $7}')  
if [ $REMAINING\_MEM -gt 4000 ]; then # If more than 4GB still available  
 log "Starting AI Agents..." "important"  
 cd /opt/sutazaiapp/agents  
 nohup python agent\_manager.py > $LOG\_DIR/agents.log 2>&1 &  
 AGENTS\_PID=$!  
 log "AI Agents started with PID: $AGENTS\_PID"  
else  
 log "Insufficient memory to start AI Agents. Please free up resources." "important"  
fi  
  
# Verify services are running  
sleep 5  
log "Verifying services..." "important"  
  
# Create status file for monitoring  
cat > /opt/sutazaiapp/status.json << EOSTATUS  
{  
 "last\_update": "$(date '+%Y-%m-%d %H:%M:%S')",  
 "components": [  
 {"name": "FastAPI Backend", "pid": $BACKEND\_PID, "port": 8000},  
 {"name": "ChromaDB", "pid": $CHROMA\_PID, "port": 8100},  
 {"name": "Streamlit UI", "pid": $UI\_PID, "port": 8501}  
 ],  
 "resources": {  
 "total\_memory\_mb": $TOTAL\_MEM,  
 "available\_memory\_mb": $AVAIL\_MEM,  
 "cpu\_cores": $CPU\_CORES  
 }  
}  
EOSTATUS  
  
log "SutazAI system startup completed" "important"  
echo "SutazAI system has been started. Access the UI at http://localhost:8501"  
EOF  
  
# Make scripts executable  
chmod +x /opt/sutazaiapp/start.sh  
  
log "SutazAI system deployment completed successfully" "important"  
echo "SutazAI system has been deployed. Run './start.sh' to start the system."  
```  
  
This implementation covers the key components of the SutazAI system, optimized for the Dell PowerEdge R720 server with its hardware constraints. The code includes memory-efficient neural architecture, ethical constraints, process security, and a lightweight web interface. All components are designed to work together as an integrated system without cloud dependencies.

**User:**

Write code for - Looking at the existing codebase and your implementation plans, I've identified how to integrate these components with the transformer optimization techniques from Intel's research. The key insight from the web search results is that we need CPU-specific optimizations for your Dell PowerEdge R720 with E5-2640 processors.  
Your existing MemoryEfficientInference class and TransformerOptimizer already implement several optimization techniques mentioned in Intel's documentation, including:  
Encoder-Decoder Attention Cache Optimization - Your code already implements attention caching in the MemoryEfficientInference class 1  
Beam Search Cache Optimization - Replacing Gather\_ND with more efficient Gather\_V2 operation is already implemented in the optimizer 1  
Memory-efficient FFN - Your TransformerOptimizer already implements optimizations similar to the padding/unpadding techniques described by Intel 1  
For your CPUOptimizedTransformer implementation, I recommend focusing on the LookupFFN technique from the "LookupFFN: Making Transformers Compute-lite for CPU inference" paper 2. This approach replaces GEMM operations with memory lookups, which is particularly effective for the E5-2640 CPUs that have limited compute capabilities but reasonable memory bandwidth.  
To fully optimize for your hardware, make these key modifications:  
Set thread count to 12 (matching E5-2640's physical cores) in your optimizers  
Use 8-bit quantization for memory efficiency  
Implement BetterTransformer's fastpath execution as already configured in your scripts  
Add the automated benchmark system from optimize\_transformers.py to your CPU-optimized architecture  
Your implementation already addresses Intel's best practices for CPU-bound transformer infer  
  
SutazAI Implementation Core Components  
I'll focus on the most critical code components from the blueprint, optimized for the Dell PowerEdge R720 server constraints.  
  
1. CPU-Optimized Neural Architecture  
python  
Copy Code  
# core/neural/cpu\_optimized\_transformer.py  
import os  
import numpy as np  
import torch  
from concurrent.futures import ThreadPoolExecutor  
from typing import Dict, List, Optional, Union  
from transformers import AutoModelForCausalLM, AutoTokenizer  
  
class CPUOptimizedTransformer:  
 def \_\_init\_\_(self, model\_path: str, quantize\_bits: int = 8):  
 """Initialize a CPU-optimized transformer model with quantization.  
   
 Args:  
 model\_path: Path to the model weights  
 quantize\_bits: Model quantization bit depth (4 or 8)  
 """  
 self.model\_path = model\_path  
 self.quantize\_bits = quantize\_bits  
 self.model = None  
 self.tokenizer = None  
 self.thread\_pool = ThreadPoolExecutor(max\_workers=os.cpu\_count())  
 self.\_load\_model()  
   
 def \_load\_model(self):  
 """Load the model with appropriate quantization."""  
 load\_kwargs = {  
 "device\_map": "auto",  
 "torch\_dtype": torch.float16,  
 }  
   
 if self.quantize\_bits == 8:  
 load\_kwargs["load\_in\_8bit"] = True  
 elif self.quantize\_bits == 4:  
 load\_kwargs["load\_in\_4bit"] = True  
 load\_kwargs["bnb\_4bit\_compute\_dtype"] = torch.float16  
 load\_kwargs["bnb\_4bit\_use\_double\_quant"] = True  
   
 self.model = AutoModelForCausalLM.from\_pretrained(  
 self.model\_path,   
 \*\*load\_kwargs  
 )  
 self.tokenizer = AutoTokenizer.from\_pretrained(self.model\_path)  
   
 def \_determine\_optimal\_batch\_size(self) -> int:  
 """Determine optimal batch size based on CPU cache and memory."""  
 # Simple heuristic based on available CPU cores and memory  
 cores = os.cpu\_count()  
 if cores <= 4:  
 return 1  
 elif cores <= 8:  
 return 2  
 else:  
 return 4  
   
 def generate(self, prompt: str, max\_tokens: int = 512) -> str:  
 """Generate text with optimized CPU usage  
   
 Args:  
 prompt: Input text prompt  
 max\_tokens: Maximum number of tokens to generate  
   
 Returns:  
 Generated text response  
 """  
 # Determine optimal batch size for CPU  
 batch\_size = self.\_determine\_optimal\_batch\_size()  
   
 # Submit to thread pool to avoid blocking  
 future = self.thread\_pool.submit(  
 self.\_generate\_with\_batching,  
 prompt,  
 max\_tokens,  
 batch\_size  
 )  
 return future.result()  
   
 def \_generate\_with\_batching(self, prompt: str, max\_tokens: int, batch\_size: int) -> str:  
 """Internal method to perform generation with batching  
   
 Args:  
 prompt: Input text prompt  
 max\_tokens: Maximum tokens to generate  
 batch\_size: Batch size for efficient CPU usage  
 """  
 # Tokenize input  
 input\_ids = self.tokenizer.encode(prompt, return\_tensors="pt")  
   
 # Generate with chunking for memory efficiency  
 generated\_tokens = []  
 chunk\_size = min(max\_tokens, 128) # Process in smaller chunks  
 remaining = max\_tokens  
   
 while remaining > 0:  
 current\_chunk = min(chunk\_size, remaining)  
 outputs = self.model.generate(  
 input\_ids,  
 max\_new\_tokens=current\_chunk,  
 do\_sample=True,  
 temperature=0.7,  
 batch\_size=batch\_size,  
 pad\_token\_id=self.tokenizer.eos\_token\_id  
 )  
   
 # Extract new tokens  
 new\_tokens = outputs[0][input\_ids.shape[1]:]  
 generated\_tokens.append(new\_tokens)  
   
 # Update for next iteration  
 input\_ids = outputs  
 remaining -= current\_chunk  
   
 # Combine results and decode  
 return self.tokenizer.decode(torch.cat(generated\_tokens), skip\_special\_tokens=True)  
2. Virtual Ethical Constraints System  
python  
Copy Code  
# core/ethics/constraint\_system.py  
import time  
import json  
import hashlib  
from typing import Dict, List, Tuple, Any  
from dataclasses import dataclass  
  
@dataclass  
class VerificationResult:  
 compliant: bool  
 reason: str = ""  
 details: Dict[str, Any] = None  
  
class VirtualEthicalConstraints:  
 def \_\_init\_\_(self, constraints\_path: str):  
 """Initialize the ethical constraint system  
   
 Args:  
 constraints\_path: Path to constraint definitions  
 """  
 self.constraints = self.\_load\_constraints(constraints\_path)  
 self.verification\_engine = self.\_initialize\_verifier()  
 self.decision\_log = []  
 self.log\_hash\_chain = self.\_initialize\_hash\_chain()  
   
 def \_load\_constraints(self, path: str) -> List[Dict]:  
 """Load constraint definitions from JSON file  
   
 Args:  
 path: Path to constraint file  
   
 Returns:  
 List of constraint definitions  
 """  
 with open(path, 'r') as f:  
 constraints = json.load(f)  
   
 # Validate constraint format  
 for constraint in constraints:  
 if not all(k in constraint for k in ['id', 'description', 'condition', 'severity']):  
 raise ValueError(f"Invalid constraint format: {constraint}")  
   
 return constraints  
   
 def \_initialize\_verifier(self):  
 """Initialize the verification engine  
   
 Returns:  
 Verification engine object  
 """  
 # This would be a more complex implementation in a real system  
 return ConstraintVerificationEngine()  
   
 def \_initialize\_hash\_chain(self) -> str:  
 """Initialize tamper-evident hash chain for logging  
   
 Returns:  
 Initial hash value  
 """  
 # Start with a hash of the constraints themselves for integrity  
 constraints\_str = json.dumps(self.constraints, sort\_keys=True)  
 return hashlib.sha256(constraints\_str.encode()).hexdigest()  
   
 def verify\_action(self, proposed\_action: Dict, context: Dict) -> Tuple[bool, str]:  
 """Verify if a proposed action complies with ethical constraints  
   
 Args:  
 proposed\_action: Action details   
 context: Contextual information  
   
 Returns:  
 Tuple of (compliant, reason)  
 """  
 # Check each constraint  
 for constraint in self.constraints:  
 verification\_result = self.verification\_engine.verify(  
 constraint=constraint,  
 action=proposed\_action,  
 context=context  
 )  
   
 if not verification\_result.compliant:  
 # Log the rejected action with hash chaining  
 log\_entry = {  
 "timestamp": time.time(),  
 "action": proposed\_action,  
 "constraint": constraint,  
 "result": "rejected",  
 "reason": verification\_result.reason,  
 "previous\_hash": self.log\_hash\_chain  
 }  
 entry\_str = json.dumps(log\_entry, sort\_keys=True)  
 log\_hash = hashlib.sha256(entry\_str.encode()).hexdigest()  
 self.log\_hash\_chain = log\_hash  
 log\_entry["hash"] = log\_hash  
 self.decision\_log.append(log\_entry)  
   
 return False, verification\_result.reason  
   
 # Action passed all constraints  
 log\_entry = {  
 "timestamp": time.time(),  
 "action": proposed\_action,  
 "result": "approved",  
 "previous\_hash": self.log\_hash\_chain  
 }  
 entry\_str = json.dumps(log\_entry, sort\_keys=True)  
 log\_hash = hashlib.sha256(entry\_str.encode()).hexdigest()  
 self.log\_hash\_chain = log\_hash  
 log\_entry["hash"] = log\_hash  
 self.decision\_log.append(log\_entry)  
   
 return True, "Action complies with all ethical constraints"  
   
 def export\_decision\_log(self, output\_path: str) -> bool:  
 """Export the tamper-evident decision log  
   
 Args:  
 output\_path: Path to save the log  
   
 Returns:  
 Success status  
 """  
 try:  
 with open(output\_path, 'w') as f:  
 json.dump(self.decision\_log, f, indent=2)  
 return True  
 except Exception as e:  
 print(f"Error exporting decision log: {e}")  
 return False  
  
class ConstraintVerificationEngine:  
 """Engine for verifying actions against constraints"""  
   
 def verify(self, constraint: Dict, action: Dict, context: Dict) -> VerificationResult:  
 """Verify an action against a constraint  
   
 Args:  
 constraint: Constraint definition  
 action: Proposed action  
 context: Context information  
   
 Returns:  
 VerificationResult with compliance status  
 """  
 # This would be a more complex implementation in a real system  
 # For demonstration, we'll implement a simple rule-based check  
 condition = constraint['condition']  
   
 # Example implementation for a few condition types  
 if condition['type'] == 'content\_safety':  
 return self.\_check\_content\_safety(action, condition)  
 elif condition['type'] == 'resource\_limit':  
 return self.\_check\_resource\_limits(action, context, condition)  
 elif condition['type'] == 'access\_control':  
 return self.\_check\_access\_control(action, context, condition)  
 else:  
 # Unknown constraint type  
 return VerificationResult(  
 compliant=False,  
 reason=f"Unknown constraint type: {condition['type']}"  
 )  
   
 def \_check\_content\_safety(self, action: Dict, condition: Dict) -> VerificationResult:  
 """Check if content meets safety requirements"""  
 if 'content' not in action:  
 return VerificationResult(compliant=True)  
   
 content = action['content']  
 prohibited\_terms = condition.get('prohibited\_terms', [])  
   
 # Simple check for prohibited terms  
 for term in prohibited\_terms:  
 if term.lower() in content.lower():  
 return VerificationResult(  
 compliant=False,  
 reason=f"Content contains prohibited term: {term}",  
 details={"term": term, "content": content}  
 )  
   
 return VerificationResult(compliant=True)  
   
 def \_check\_resource\_limits(self, action: Dict, context: Dict, condition: Dict) -> VerificationResult:  
 """Check if action exceeds resource limits"""  
 if 'resource\_usage' not in action:  
 return VerificationResult(compliant=True)  
   
 usage = action['resource\_usage']  
 limits = condition.get('limits', {})  
   
 # Check each resource limit  
 for resource, limit in limits.items():  
 if resource in usage and usage[resource] > limit:  
 return VerificationResult(  
 compliant=False,  
 reason=f"Exceeds {resource} limit of {limit}",  
 details={"resource": resource, "usage": usage[resource], "limit": limit}  
 )  
   
 return VerificationResult(compliant=True)  
   
 def \_check\_access\_control(self, action: Dict, context: Dict, condition: Dict) -> VerificationResult:  
 """Check if action has proper authorization"""  
 if 'actor' not in context:  
 return VerificationResult(  
 compliant=False,  
 reason="Missing actor information in context"  
 )  
   
 actor = context['actor']  
 required\_role = condition.get('required\_role')  
   
 if required\_role and ('role' not in actor or actor['role'] != required\_role):  
 return VerificationResult(  
 compliant=False,  
 reason=f"Requires role {required\_role}",  
 details={"actor": actor, "required\_role": required\_role}  
 )  
   
 return VerificationResult(compliant=True)  
3. Memory-Efficient Inference Engine  
python  
Copy Code  
# core/inference/memory\_efficient\_inference.py  
import os  
import time  
import psutil  
import torch  
from typing import Dict, List, Any, Optional  
from threading import Lock  
  
class MemoryEfficientInference:  
 def \_\_init\_\_(self, model\_manager):  
 """Initialize memory-efficient inference system  
   
 Args:  
 model\_manager: Model loading/unloading manager  
 """  
 self.model\_manager = model\_manager  
 self.active\_models = {}  
 self.memory\_monitor = MemoryMonitor()  
 self.model\_lock = Lock()  
   
 def run\_inference(self, model\_name: str, input\_data: Dict, priority: str = "normal") -> Dict:  
 """Run inference with efficient memory management  
   
 Args:  
 model\_name: Name of model to use  
 input\_data: Input data for inference  
 priority: Task priority ("high", "normal", "low")  
   
 Returns:  
 Inference results  
 """  
 # Check if we have enough memory to load the model  
 required\_memory = self.model\_manager.get\_model\_memory\_requirement(model\_name)  
 available\_memory = self.memory\_monitor.get\_available\_memory()  
   
 with self.model\_lock:  
 if required\_memory > available\_memory:  
 # Not enough memory - unload less important models  
 self.\_free\_memory\_for\_model(required\_memory)  
   
 # Load model if not already loaded  
 if model\_name not in self.active\_models:  
 self.active\_models[model\_name] = self.model\_manager.load\_model(  
 model\_name,   
 quantize=True  
 )  
   
 # Mark model as in use  
 self.active\_models[model\_name]["in\_use"] = True  
 self.active\_models[model\_name]["last\_used"] = time.time()  
 self.active\_models[model\_name]["priority"] = priority  
   
 try:  
 # Run inference with appropriate priority  
 result = self.\_prioritized\_inference(  
 model\_name,  
 input\_data,  
 priority  
 )  
   
 return result  
 finally:  
 # Mark model as no longer in use  
 with self.model\_lock:  
 if model\_name in self.active\_models:  
 self.active\_models[model\_name]["in\_use"] = False  
   
 # Consider unloading models based on memory pressure  
 self.\_consider\_unloading\_models()  
   
 def \_free\_memory\_for\_model(self, required\_memory: int) -> None:  
 """Unload models to free memory  
   
 Args:  
 required\_memory: Amount of memory needed in MB  
 """  
 # Sort models based on priority and last used time  
 models\_to\_unload = []  
   
 for model\_name, model\_info in self.active\_models.items():  
 if not model\_info["in\_use"]:  
 models\_to\_unload.append((  
 model\_name,  
 model\_info["priority"],  
 model\_info["last\_used"]  
 ))  
   
 # Sort by priority (low first) and then by last\_used (oldest first)  
 models\_to\_unload.sort(key=lambda x: (  
 0 if x[1] == "low" else 1 if x[1] == "normal" else 2,  
 x[2]  
 ))  
   
 # Unload models until we have enough memory  
 for model\_name, \_, \_ in models\_to\_unload:  
 self.model\_manager.unload\_model(model\_name)  
 del self.active\_models[model\_name]  
   
 # Check if we now have enough memory  
 if self.memory\_monitor.get\_available\_memory() >= required\_memory:  
 break  
   
 def \_prioritized\_inference(self, model\_name: str, input\_data: Dict, priority: str) -> Dict:  
 """Run inference with priority-based resource allocation  
   
 Args:  
 model\_name: Model to use  
 input\_data: Input data  
 priority: Priority level  
   
 Returns:  
 Inference results  
 """  
 model = self.active\_models[model\_name]["model"]  
   
 # Set thread priority based on task priority  
 if priority == "high":  
 # Use more CPU cores for high priority tasks  
 n\_threads = min(os.cpu\_count(), 8)  
 torch.set\_num\_threads(n\_threads)  
 elif priority == "normal":  
 # Use moderate CPU cores for normal priority  
 n\_threads = min(os.cpu\_count() // 2, 4)  
 torch.set\_num\_threads(n\_threads)  
 else: # low priority  
 # Use minimal CPU cores for low priority  
 torch.set\_num\_threads(2)  
   
 # Process input in streaming fashion for memory efficiency  
 result = self.\_stream\_inference(model, input\_data)  
   
 # Reset thread count after completion  
 torch.set\_num\_threads(1)  
   
 return result  
   
 def \_stream\_inference(self, model, input\_data: Dict) -> Dict:  
 """Process data in a streaming fashion to minimize memory usage  
   
 Args:  
 model: The loaded model  
 input\_data: Input data for inference  
   
 Returns:  
 Processed results  
 """  
 # This would implement streaming processing based on model type  
 # For this example, we'll provide a simplified implementation  
   
 # Handle text data with chunking if too large  
 if "text" in input\_data and len(input\_data["text"]) > 10000:  
 chunks = self.\_split\_text\_into\_chunks(input\_data["text"], chunk\_size=4000)  
 results = []  
   
 for chunk in chunks:  
 # Process each chunk separately  
 chunk\_result = model({"text": chunk})  
 results.append(chunk\_result)  
   
 # Combine results  
 return self.\_combine\_chunked\_results(results)  
   
 # Standard processing for smaller inputs  
 return model(input\_data)  
   
 def \_split\_text\_into\_chunks(self, text: str, chunk\_size: int) -> List[str]:  
 """Split text into manageable chunks while preserving context  
   
 Args:  
 text: Text to split  
 chunk\_size: Target chunk size  
   
 Returns:  
 List of text chunks  
 """  
 chunks = []  
   
 # Split by paragraphs first  
 paragraphs = text.split("\n\n")  
 current\_chunk = []  
 current\_size = 0  
   
 for para in paragraphs:  
 para\_size = len(para)  
   
 # If paragraph fits in current chunk, add it  
 if current\_size + para\_size <= chunk\_size:  
 current\_chunk.append(para)  
 current\_size += para\_size  
 # If paragraph is too big by itself, split it  
 elif para\_size > chunk\_size:  
 # If we have content in current chunk, finalize it  
 if current\_chunk:  
 chunks.append("\n\n".join(current\_chunk))  
 current\_chunk = []  
 current\_size = 0  
   
 # Split the paragraph  
 sentences = para.split(". ")  
 sentences = [s + "." if not s.endswith(".") else s for s in sentences]  
   
 sent\_chunk = []  
 sent\_size = 0  
   
 for sent in sentences:  
 sent\_size = len(sent)  
   
 if sent\_size + sent\_size <= chunk\_size:  
 sent\_chunk.append(sent)  
 sent\_size += sent\_size  
 else:  
 # If sentence is too big, we have to split words  
 if sent\_chunk:  
 chunks.append(" ".join(sent\_chunk))  
 sent\_chunk = []  
 sent\_size = 0  
   
 # Add the sentence as its own chunk  
 chunks.append(sent)  
   
 if sent\_chunk:  
 chunks.append(" ".join(sent\_chunk))  
 # If adding paragraph exceeds chunk size, finalize current and start new  
 else:  
 chunks.append("\n\n".join(current\_chunk))  
 current\_chunk = [para]  
 current\_size = para\_size  
   
 # Add final chunk if there's content  
 if current\_chunk:  
 chunks.append("\n\n".join(current\_chunk))  
   
 return chunks  
   
 def \_combine\_chunked\_results(self, results: List[Dict]) -> Dict:  
 """Combine results from chunked processing  
   
 Args:  
 results: List of chunk results  
   
 Returns:  
 Combined result  
 """  
 # This would be specific to the model output format  
 # For this example, let's assume a simple case with text outputs  
   
 if all("text" in r for r in results):  
 combined\_text = " ".join(r["text"] for r in results)  
 return {"text": combined\_text}  
   
 # For other types, we'd need specific combination logic  
 # Just return the first result as fallback  
 return results[0] if results else {}  
   
 def \_consider\_unloading\_models(self) -> None:  
 """Consider unloading inactive models based on memory pressure"""  
 # Check current memory pressure  
 memory\_pressure = self.memory\_monitor.get\_memory\_pressure()  
   
 # High memory pressure: unload all inactive models  
 if memory\_pressure > 0.8: # > 80% memory used  
 for model\_name, model\_info in list(self.active\_models.items()):  
 if not model\_info["in\_use"]:  
 self.model\_manager.unload\_model(model\_name)  
 del self.active\_models[model\_name]  
   
 # Medium memory pressure: unload low priority inactive models  
 elif memory\_pressure > 0.6: # > 60% memory used  
 for model\_name, model\_info in list(self.active\_models.items()):  
 if not model\_info["in\_use"] and model\_info["priority"] == "low":  
 last\_used\_time = model\_info["last\_used"]  
 current\_time = time.time()  
   
 # If not used in the last 10 minutes  
 if current\_time - last\_used\_time > 600:  
 self.model\_manager.unload\_model(model\_name)  
 del self.active\_models[model\_name]  
  
  
class MemoryMonitor:  
 """Monitor system memory for resource management"""  
   
 def \_\_init\_\_(self):  
 self.last\_update = 0  
 self.cached\_stats = None  
 self.update\_interval = 5 # seconds  
   
 def get\_available\_memory(self) -> int:  
 """Get available system memory in MB  
   
 Returns:  
 Available memory in MB  
 """  
 self.\_update\_stats\_if\_needed()  
 return self.cached\_stats["available\_mb"]  
   
 def get\_memory\_pressure(self) -> float:  
 """Get memory pressure as percentage used  
   
 Returns:  
 Memory pressure (0.0 to 1.0)  
 """  
 self.\_update\_stats\_if\_needed()  
 return self.cached\_stats["pressure"]  
   
 def \_update\_stats\_if\_needed(self) -> None:  
 """Update memory stats if cache is stale"""  
 current\_time = time.time()  
   
 if current\_time - self.last\_update > self.update\_interval:  
 self.\_update\_stats()  
 self.last\_update = current\_time  
   
 def \_update\_stats(self) -> None:  
 """Update memory statistics"""  
 mem = psutil.virtual\_memory()  
   
 self.cached\_stats = {  
 "total\_mb": mem.total // (1024 \* 1024),  
 "available\_mb": mem.available // (1024 \* 1024),  
 "used\_mb": (mem.total - mem.available) // (1024 \* 1024),  
 "pressure": 1.0 - (mem.available / mem.total)  
 }  
4. Process Security Manager  
python  
Copy Code  
# core/security/process\_security.py  
import uuid  
import time  
import subprocess  
import os  
import json  
import psutil  
from typing import Dict, List, Optional, Tuple  
  
class ProcessSecurityManager:  
 def \_\_init\_\_(self, log\_dir: str = "/opt/sutazaiapp/logs"):  
 """Initialize the process security manager  
   
 Args:  
 log\_dir: Directory for security logs  
 """  
 self.secured\_processes = {}  
 self.log\_dir = log\_dir  
 self.security\_profiles = self.\_load\_security\_profiles()  
   
 # Ensure log directory exists  
 os.makedirs(log\_dir, exist\_ok=True)  
   
 def \_load\_security\_profiles(self) -> Dict:  
 """Load security profiles from configuration  
   
 Returns:  
 Dict of security profiles  
 """  
 # In a real system, this would load from a config file  
 # Here we define some default profiles  
 return {  
 "high\_security": {  
 "namespace": ["--private-network", "--private-tmp"],  
 "cgroup": ["--memory-max=2G", "--cpu-quota=50%"],  
 "capabilities": ["--drop-capability=all", "--add-capability=NET\_BIND\_SERVICE"]  
 },  
 "medium\_security": {  
 "namespace": ["--private-tmp"],  
 "cgroup": ["--memory-max=4G", "--cpu-quota=80%"],  
 "capabilities": ["--drop-capability=CAP\_SYS\_ADMIN", "--drop-capability=CAP\_NET\_ADMIN"]  
 },  
 "low\_security": {  
 "namespace": [],  
 "cgroup": ["--memory-max=8G"],  
 "capabilities": ["--drop-capability=CAP\_SYS\_ADMIN"]  
 }  
 }  
   
 def launch\_secured\_process(self, executable\_path: str, args: List[str],   
 security\_profile: str = "medium\_security") -> str:  
 """Launch a process with security controls  
   
 Args:  
 executable\_path: Path to executable  
 args: Command arguments  
 security\_profile: Security profile to apply  
   
 Returns:  
 Process ID  
 """  
 # Generate unique ID for this process  
 process\_id = str(uuid.uuid4())  
   
 # Get security configuration  
 if security\_profile not in self.security\_profiles:  
 raise ValueError(f"Unknown security profile: {security\_profile}")  
   
 profile = self.security\_profiles[security\_profile]  
   
 # Build the systemd-run command with security controls  
 namespace\_config = profile.get("namespace", [])  
 cgroup\_config = profile.get("cgroup", [])  
 capability\_config = profile.get("capabilities", [])  
   
 # Create command  
 cmd = ["systemd-run", "--scope", f"--unit=sutazai-{process\_id}"]  
 cmd.extend(namespace\_config)  
 cmd.extend(cgroup\_config)  
 cmd.extend(capability\_config)  
 cmd.append(executable\_path)  
 cmd.extend(args)  
   
 # Log the process launch  
 self.\_log\_security\_event(process\_id, "process\_launch", {  
 "executable": executable\_path,  
 "args": args,  
 "security\_profile": security\_profile,  
 "command": " ".join(cmd)  
 })  
   
 # Launch process  
 try:  
 process = subprocess.Popen(  
 cmd,  
 stdout=subprocess.PIPE,  
 stderr=subprocess.PIPE  
 )  
   
 # Store process information  
 self.secured\_processes[process\_id] = {  
 "process": process,  
 "executable": executable\_path,  
 "args": args,  
 "security\_profile": security\_profile,  
 "start\_time": time.time(),  
 "last\_checked": time.time(),  
 "anomalies": []  
 }  
   
 return process\_id  
   
 except Exception as e:  
 self.\_log\_security\_event(process\_id, "process\_launch\_failed", {  
 "error": str(e)  
 })  
 raise  
   
 def monitor\_process(self, process\_id: str) -> Optional[Dict]:  
 """Monitor a secured process for anomalies  
   
 Args:  
 process\_id: Process ID to monitor  
   
 Returns:  
 Process status information  
 """  
 if process\_id not in self.secured\_processes:  
 return None  
   
 process\_info = self.secured\_processes[process\_id]  
 process = process\_info["process"]  
   
 # Check if process is still running  
 if process.poll() is not None:  
 # Process has terminated  
 return {  
 "process\_id": process\_id,  
 "status": "terminated",  
 "exit\_code": process.returncode,  
 "runtime": time.time() - process\_info["start\_time"]  
 }  
   
 # Get resource usage  
 try:  
 proc = psutil.Process(process.pid)  
   
 # Get resource metrics  
 cpu\_percent = proc.cpu\_percent(interval=0.1)  
 memory\_info = proc.memory\_info()  
 io\_counters = proc.io\_counters() if hasattr(proc, "io\_counters") else None  
   
 resource\_usage = {  
 "cpu\_percent": cpu\_percent,  
 "memory\_mb": memory\_info.rss // (1024 \* 1024),  
 "io\_read\_mb": io\_counters.read\_bytes // (1024 \* 1024) if io\_counters else 0,  
 "io\_write\_mb": io\_counters.write\_bytes // (1024 \* 1024) if io\_counters else 0  
 }  
   
 # Detect anomalies  
 anomalies = self.\_detect\_anomalies(process\_id, resource\_usage)  
 if anomalies:  
 process\_info["anomalies"].extend(anomalies)  
   
 # Log anomalies  
 for anomaly in anomalies:  
 self.\_log\_security\_event(process\_id, "process\_anomaly", anomaly)  
   
 # Update last checked time  
 process\_info["last\_checked"] = time.time()  
   
 return {  
 "process\_id": process\_id,  
 "status": "running",  
 "resource\_usage": resource\_usage,  
 "runtime": time.time() - process\_info["start\_time"],  
 "anomalies": process\_info["anomalies"]  
 }  
   
 except (psutil.NoSuchProcess, psutil.AccessDenied) as e:  
 self.\_log\_security\_event(process\_id, "process\_monitoring\_error", {  
 "error": str(e)  
 })  
 return {  
 "process\_id": process\_id,  
 "status": "error",  
 "error": str(e)  
 }  
   
 def terminate\_process(self, process\_id: str, reason: str = "manual") -> bool:  
 """Terminate a secured process  
   
 Args:  
 process\_id: Process ID to terminate  
 reason: Reason for termination  
   
 Returns:  
 Success status  
 """  
 if process\_id not in self.secured\_processes:  
 return False  
   
 process\_info = self.secured\_processes[process\_id]  
 process = process\_info["process"]  
   
 # Log termination  
 self.\_log\_security\_event(process\_id, "process\_termination", {  
 "reason": reason,  
 "runtime": time.time() - process\_info["start\_time"]  
 })  
   
 # Terminate the process  
 try:  
 process.terminate()  
 # Wait for process to exit (timeout after 5 seconds)  
 try:  
 process.wait(timeout=5)  
 except subprocess.TimeoutExpired:  
 # Force kill if it doesn't exit cleanly  
 process.kill()  
 process.wait()  
   
 return True  
 except Exception as e:  
 self.\_log\_security\_event(process\_id, "process\_termination\_error", {  
 "error": str(e)  
 })  
 return False  
   
 def \_detect\_anomalies(self, process\_id: str, resource\_usage: Dict) -> List[Dict]:  
 """Detect anomalies in process behavior  
   
 Args:  
 process\_id: Process ID  
 resource\_usage: Current resource usage  
   
 Returns:  
 List of detected anomalies  
 """  
 anomalies = []  
 process\_info = self.secured\_processes[process\_id]  
 security\_profile = process\_info["security\_profile"]  
 profile = self.security\_profiles[security\_profile]  
   
 # Check CPU usage  
 if resource\_usage["cpu\_percent"] > 95: # Over 95% CPU  
 anomalies.append({  
 "type": "high\_cpu",  
 "value": resource\_usage["cpu\_percent"],  
 "timestamp": time.time()  
 })  
   
 # Check memory limits from cgroups  
 for cgroup\_option in profile.get("cgroup", []):  
 if cgroup\_option.startswith("--memory-max="):  
 limit\_str = cgroup\_option.split("=")[1]  
 limit\_mb = self.\_parse\_memory\_limit(limit\_str)  
   
 if limit\_mb and resource\_usage["memory\_mb"] > limit\_mb \* 0.9: # Over 90% of limit  
 anomalies.append({  
 "type": "high\_memory",  
 "value": resource\_usage["memory\_mb"],  
 "limit": limit\_mb,  
 "timestamp": time.time()  
 })  
   
 # Check IO operations (could indicate data exfiltration)  
 if hasattr(process\_info, "last\_io\_write") and resource\_usage["io\_write\_mb"] - process\_info.get("last\_io\_write", 0) > 500:  
 # More than 500MB written since last check  
 anomalies.append({  
 "type": "high\_io\_write",  
 "value": resource\_usage["io\_write\_mb"] - process\_info.get("last\_io\_write", 0),  
 "timestamp": time.time()  
 })  
   
 # Store current IO values for next comparison  
 process\_info["last\_io\_write"] = resource\_usage["io\_write\_mb"]  
 process\_info["last\_io\_read"] = resource\_usage["io\_read\_mb"]  
   
 return anomalies  
   
 def \_parse\_memory\_limit(self, limit\_str: str) -> Optional[int]:  
 """Parse memory limit string to MB  
   
 Args:  
 limit\_str: Memory limit string (e.g., "2G")  
   
 Returns:  
 Memory limit in MB or None if invalid  
 """  
 try:  
 if limit\_str.endswith("G"):  
 return int(float(limit\_str[:-1]) \* 1024)  
 elif limit\_str.endswith("M"):  
 return int(float(limit\_str[:-1]))  
 elif limit\_str.endswith("K"):  
 return int(float(limit\_str[:-1]) / 1024)  
 else:  
 # Assume bytes  
 return int(float(limit\_str) / (1024 \* 1024))  
 except ValueError:  
 return None  
   
 def \_log\_security\_event(self, process\_id: str, event\_type: str, event\_data: Dict) -> None:  
 """Log a security event  
   
 Args:  
 process\_id: Process ID  
 event\_type: Type of security event  
 event\_data: Event data to log  
 """  
 log\_entry = {  
 "timestamp": time.time(),  
 "process\_id": process\_id,  
 "event\_type": event\_type,  
 "data": event\_data  
 }  
   
 log\_file = os.path.join(self.log\_dir, "security\_events.log")  
 with open(log\_file, "a") as f:  
 f.write(json.dumps(log\_entry) + "\n")  
5. FastAPI Backend (Resource-Efficient)  
python  
Copy Code  
# api/backend/main.py  
import os  
import logging  
import psutil  
import time  
from typing import Dict, List, Optional  
from fastapi import FastAPI, BackgroundTasks, Depends, HTTPException, Request  
from fastapi.middleware.cors import CORSMiddleware  
from pydantic import BaseModel  
from concurrent.futures import ThreadPoolExecutor  
  
# Setup logging to file to avoid console I/O overhead  
logging.basicConfig(  
 level=logging.INFO,  
 filename="/opt/sutazaiapp/logs/backend.log",  
 format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'  
)  
logger = logging.getLogger("sutazai-backend")  
  
# Initialize FastAPI app  
app = FastAPI(title="SutazAI Backend")  
  
# Cache values  
\_cache = {  
 "cpu\_usage": 0,  
 "memory\_usage": 0,  
 "component\_status": [],  
 "last\_updated": 0  
}  
  
# Request models  
class InferenceRequest(BaseModel):  
 input\_text: str  
 model\_name: str = "default"  
 max\_tokens: int = 512  
 expected\_duration: int = 5 # Expected seconds to complete  
 priority: str = "normal" # "high", "normal", "low"  
  
class AgentTaskRequest(BaseModel):  
 task\_description: str  
 agent\_type: str  
 priority: str = "normal"  
 timeout: int = 300 # 5 minutes  
  
# Response models  
class SystemStatusResponse(BaseModel):  
 status: str  
 cpu\_usage: float  
 memory\_usage: float  
 components: List[Dict]  
  
class InferenceResponse(BaseModel):  
 task\_id: Optional[str] = None  
 status: str  
 result: Optional[str] = None  
 error: Optional[str] = None  
  
# Add CORS middleware to reduce overhead  
app.add\_middleware(  
 CORSMiddleware,  
 allow\_origins=["http://localhost:8501"], # Streamlit UI only  
 allow\_credentials=True,  
 allow\_methods=["GET", "POST"],  
 allow\_headers=["\*"],  
)  
  
# Global variables  
thread\_pool = None  
db\_pool = None  
  
@app.on\_event("startup")  
async def startup\_event():  
 """Initialize resources on startup"""  
 global thread\_pool  
 # Initialize thread pool for CPU-bound tasks  
 cpu\_count = psutil.cpu\_count()  
 max\_workers = max(2, cpu\_count - 1) # Leave at least one core free  
 thread\_pool = ThreadPoolExecutor(max\_workers=max\_workers)  
   
 logger.info(f"Backend started with {max\_workers} worker threads")  
   
 # Start background thread for periodic status updates  
 thread\_pool.submit(update\_status\_periodically)  
  
def update\_status\_periodically():  
 """Update cached status values every 5 seconds"""  
 try:  
 while True:  
 # Update CPU usage (average over 1 second)  
 \_cache["cpu\_usage"] = psutil.cpu\_percent(interval=1)  
   
 # Update memory usage  
 mem = psutil.virtual\_memory()  
 \_cache["memory\_usage"] = mem.percent  
   
 # Update component status (simplified example)  
 \_cache["component\_status"] = [  
 {"name": "FastAPI Backend", "status": "running"},  
 {"name": "Database", "status": "running"},  
 {"name": "AI Engine", "status": "running"}  
 ]  
   
 \_cache["last\_updated"] = time.time()  
   
 # Sleep for 4 seconds (plus 1 second from cpu\_percent)  
 time.sleep(4)  
 except Exception as e:  
 logger.error(f"Error in status update thread: {e}")  
  
@app.get("/system/status", response\_model=SystemStatusResponse)  
async def get\_system\_status():  
 """Get system status with minimal resource usage"""  
 # Use cached values for CPU/memory  
 return {  
 "status": "Running",  
 "cpu\_usage": \_cache["cpu\_usage"],  
 "memory\_usage": \_cache["memory\_usage"],  
 "components": \_cache["component\_status"]  
 }  
  
def has\_sufficient\_memory(request: InferenceRequest) -> bool:  
 """Check if there's enough memory for the request  
   
 Args:  
 request: The inference request  
   
 Returns:  
 True if sufficient memory is available  
 """  
 # Simple heuristic based on model and request size  
 mem = psutil.virtual\_memory()  
 available\_mb = mem.available // (1024 \* 1024)  
   
 # Estimate memory needed (very rough estimate)  
 if request.model\_name == "large":  
 needed\_mb = 4000 # 4GB  
 elif request.model\_name == "medium":  
 needed\_mb = 2000 # 2GB  
 else:  
 needed\_mb = 1000 # 1GB  
   
 # Add overhead for input size  
 input\_size\_mb = len(request.input\_text) // (1024 \* 1024) + 1  
 needed\_mb += input\_size\_mb \* 2 # Double for processing overhead  
   
 logger.info(f"Memory check: available={available\_mb}MB, needed={needed\_mb}MB")  
 return available\_mb >= needed\_mb  
  
def generate\_task\_id() -> str:  
 """Generate a unique task ID  
   
 Returns:  
 Task ID string  
 """  
 import uuid  
 return str(uuid.uuid4())  
  
async def run\_inference\_sync(request: InferenceRequest) -> Dict:  
 """Run inference synchronously  
   
 Args:  
 request: Inference request  
   
 Returns:  
 Inference results  
 """  
 try:  
 # This would call the actual inference engine  
 # Simplified implementation for example  
 import time  
   
 # Simulate varying processing time based on model size  
 if request.model\_name == "large":  
 time.sleep(1.0)  
 elif request.model\_name == "medium":  
 time.sleep(0.5)  
 else:  
 time.sleep(0.2)  
   
 # Return simulated result  
 return {  
 "status": "completed",  
 "result": f"Response to: {request.input\_text[:50]}..."  
 }  
 except Exception as e:  
 logger.error(f"Error in inference: {e}")  
 return {  
 "status": "error",  
 "error": str(e)  
 }  
  
def run\_inference\_task(request: InferenceRequest, task\_id: str):  
 """Run inference as background task  
   
 Args:  
 request: Inference request  
 task\_id: Unique task ID  
 """  
 try:  
 # This would call the actual inference engine  
 # Simplified implementation for example  
 import time  
   
 # Store task start in database  
 # db\_pool.execute("INSERT INTO tasks (id, status) VALUES (?, ?)",   
 # (task\_id, "processing"))  
   
 # Simulate processing time  
 if request.model\_name == "large":  
 time.sleep(3.0)  
 elif request.model\_name == "medium":  
 time.sleep(1.5)  
 else:  
 time.sleep(0.5)  
   
 # Store result in database  
 result = f"Response to: {request.input\_text[:50]}..."  
 # db\_pool.execute("UPDATE tasks SET status = ?, result = ? WHERE id = ?",   
 # ("completed", result, task\_id))  
   
 except Exception as e:  
 logger.error(f"Error in background inference task: {e}")  
 # Store error in database  
 # db\_pool.execute("UPDATE tasks SET status = ?, error = ? WHERE id = ?",   
 # ("error", str(e), task\_id))  
  
@app.post("/inference", response\_model=InferenceResponse)  
async def run\_inference(request: InferenceRequest, background\_tasks: BackgroundTasks):  
 """Run inference with resource management  
   
 Args:  
 request: Inference parameters  
 background\_tasks: FastAPI background tasks handler  
   
 Returns:  
 Inference result or task ID  
 """  
 # Check available memory before accepting large requests  
 if not has\_sufficient\_memory(request):  
 raise HTTPException(status\_code=503, detail="Insufficient system resources")  
   
 # For long-running inference, use background task  
 if request.expected\_duration > 5:  
 task\_id = generate\_task\_id()  
 background\_tasks.add\_task(run\_inference\_task, request, task\_id)  
 return {"task\_id": task\_id, "status": "processing"}  
   
 # For quick inference, run directly  
 result = await run\_inference\_sync(request)  
 return result  
  
@app.get("/task/{task\_id}", response\_model=InferenceResponse)  
async def get\_task\_status(task\_id: str):  
 """Check status of background task  
   
 Args:  
 task\_id: Task ID  
   
 Returns:  
 Task status and result if complete  
 """  
 # This would query the database for task status  
 # Simplified implementation for example  
 import random  
   
 # Simulate random task state  
 rand = random.random()  
 if rand < 0.2:  
 return {"task\_id": task\_id, "status": "processing"}  
 elif rand < 0.9:  
 return {  
 "task\_id": task\_id,   
 "status": "completed",  
 "result": f"Completed task {task\_id}"  
 }  
 else:  
 return {  
 "task\_id": task\_id,   
 "status": "error",  
 "error": "Simulated error in task processing"  
 }  
  
@app.middleware("http")  
async def add\_process\_time\_header(request: Request, call\_next):  
 """Middleware to track request processing time  
   
 Args:  
 request: HTTP request  
 call\_next: Next middleware function  
   
 Returns:  
 HTTP response  
 """  
 start\_time = time.time()  
 response = await call\_next(request)  
 process\_time = time.time() - start\_time  
   
 # Add processing time header  
 response.headers["X-Process-Time"] = str(process\_time)  
   
 # Log request (only for significant requests)  
 if process\_time > 0.5:  
 logger.info(f"Slow request: {request.url.path} took {process\_time:.2f}s")  
   
 return response  
6. Streamlit UI (Lightweight)  
python  
Copy Code  
# api/ui/app.py  
import streamlit as st  
import requests  
import json  
import time  
import os  
from datetime import datetime  
  
# Reduce Streamlit's memory footprint  
st.set\_page\_config(  
 page\_title="SutazAI Management",  
 layout="wide",  
 initial\_sidebar\_state="collapsed" # Save memory by starting collapsed  
)  
  
# Use session state sparingly to reduce memory usage  
if "authenticated" not in st.session\_state:  
 st.session\_state.authenticated = False  
if "last\_update" not in st.session\_state:  
 st.session\_state.last\_update = 0  
if "system\_status" not in st.session\_state:  
 st.session\_state.system\_status = None  
if "active\_tasks" not in st.session\_state:  
 st.session\_state.active\_tasks = {}  
  
# Constants  
API\_URL = "http://localhost:8000"  
  
# Function to efficiently fetch data with caching  
def get\_system\_status(force\_refresh=False):  
 """Get system status with caching  
   
 Args:  
 force\_refresh: Force refresh regardless of cache  
   
 Returns:  
 System status data  
 """  
 current\_time = time.time()  
 # Only refresh data every 10 seconds unless forced  
 if force\_refresh or (current\_time - st.session\_state.last\_update) > 10:  
 try:  
 response = requests.get(f"{API\_URL}/system/status", timeout=2)  
 st.session\_state.system\_status = response.json()  
 st.session\_state.last\_update = current\_time  
 except Exception as e:  
 # Don't update timestamp on error to allow retry  
 if st.session\_state.system\_status is None:  
 st.session\_state.system\_status = {"status": "Error", "error": str(e)}  
   
 return st.session\_state.system\_status  
  
def run\_inference(input\_text, model="default", max\_tokens=512):  
 """Run inference through API  
   
 Args:  
 input\_text: User input text  
 model: Model to use  
 max\_tokens: Maximum tokens to generate  
   
 Returns:  
 Inference result  
 """  
 try:  
 data = {  
 "input\_text": input\_text,  
 "model\_name": model,  
 "max\_tokens": max\_tokens,  
 "expected\_duration": 10 if model == "large" else 5  
 }  
   
 response = requests.post(f"{API\_URL}/inference", json=data, timeout=30)  
 result = response.json()  
   
 # If background task, store task ID for polling  
 if "task\_id" in result and result["status"] == "processing":  
 st.session\_state.active\_tasks[result["task\_id"]] = {  
 "start\_time": time.time(),  
 "model": model,  
 "input\_preview": input\_text[:50] + "..." if len(input\_text) > 50 else input\_text  
 }  
   
 return result  
 except Exception as e:  
 return {"status": "error", "error": str(e)}  
  
def check\_task\_status(task\_id):  
 """Check status of a background task  
   
 Args:  
 task\_id: Task ID to check  
   
 Returns:  
 Task status  
 """  
 try:  
 response = requests.get(f"{API\_URL}/task/{task\_id}", timeout=5)  
 return response.json()  
 except Exception as e:  
 return {"status": "error", "error": str(e)}  
  
def update\_active\_tasks():  
 """Update status of all active tasks"""  
 tasks\_to\_remove = []  
   
 for task\_id, task\_info in st.session\_state.active\_tasks.items():  
 result = check\_task\_status(task\_id)  
   
 # Update task info  
 st.session\_state.active\_tasks[task\_id].update(result)  
   
 # Mark completed or errored tasks for removal  
 if result["status"] in ["completed", "error"]:  
 tasks\_to\_remove.append(task\_id)  
   
 # Remove completed/errored tasks after 30 seconds  
 current\_time = time.time()  
 for task\_id in list(st.session\_state.active\_tasks.keys()):  
 task\_info = st.session\_state.active\_tasks[task\_id]  
 if task\_info["status"] in ["completed", "error"]:  
 if current\_time - task\_info.get("completion\_time", current\_time) > 30:  
 tasks\_to\_remove.append(task\_id)  
   
 # Remove tasks  
 for task\_id in tasks\_to\_remove:  
 if task\_id in st.session\_state.active\_tasks:  
 del st.session\_state.active\_tasks[task\_id]  
  
# Main application with optimized rendering  
def main():  
 """Main application logic"""  
 # Authentication with minimal overhead  
 if not st.session\_state.authenticated:  
 st.title("SutazAI Login")  
 col1, col2 = st.columns(2)  
 with col1:  
 username = st.text\_input("Username")  
 with col2:  
 password = st.text\_input("Password", type="password")  
   
 if st.button("Login"):  
 # Simple auth for demo - replace with secure auth  
 if username == "admin" and password == "sutazai":  
 st.session\_state.authenticated = True  
 st.experimental\_rerun()  
 else:  
 st.error("Invalid credentials")  
 return  
   
 # Main UI with tabbed interface to reduce rendering overhead  
 st.title("SutazAI Management Console")  
 tab1, tab2, tab3 = st.tabs(["System", "Chat", "Tasks"])  
   
 # System tab - load efficiently  
 with tab1:  
 status = get\_system\_status()  
   
 # Refresh button - only re-fetch data when clicked  
 if st.button("Refresh Status"):  
 status = get\_system\_status(force\_refresh=True)  
   
 # Display system metrics with minimal components  
 col1, col2, col3 = st.columns(3)  
 col1.metric("Status", status.get("status", "Unknown"))  
 col2.metric("CPU Usage", f"{status.get('cpu\_usage', 0):.1f}%")  
 col3.metric("Memory Usage", f"{status.get('memory\_usage', 0):.1f}%")  
   
 # Only display components if we have them  
 if "components" in status:  
 st.subheader("System Components")  
 # Use native table instead of DataFrame for efficiency  
 st.table(status["components"])  
   
 # Chat interface - optimized for minimal rendering  
 with tab2:  
 st.subheader("Chat with SutazAI")  
   
 # Model selection (simple radio to avoid heavy widgets)  
 model = st.radio(  
 "Select model:",  
 ["default", "medium", "large"],  
 horizontal=True  
 )  
   
 # Chat input  
 user\_input = st.text\_area("Enter your message:", height=100)  
   
 # Submit button  
 if st.button("Send"):  
 if user\_input:  
 with st.spinner("Processing..."):  
 result = run\_inference(user\_input, model)  
   
 # Display result  
 if result["status"] == "completed":  
 st.success("Response received")  
 st.write(result["result"])  
 elif result["status"] == "processing":  
 st.info(f"Processing as background task (ID: {result['task\_id']})")  
 else:  
 st.error(f"Error: {result.get('error', 'Unknown error')}")  
   
 # Tasks tab - show active tasks  
 with tab3:  
 st.subheader("Active Tasks")  
   
 # Update task status  
 update\_active\_tasks()  
   
 # Display active tasks  
 if st.session\_state.active\_tasks:  
 for task\_id, task\_info in st.session\_state.active\_tasks.items():  
 status = task\_info.get("status", "processing")  
   
 # Determine badge color  
 if status == "processing":  
 badge = "🔄"  
 elif status == "completed":  
 badge = "✅"  
 else:  
 badge = "❌"  
   
 # Calculate runtime  
 if "start\_time" in task\_info:  
 runtime = time.time() - task\_info["start\_time"]  
 runtime\_str = f"{runtime:.1f}s"  
 else:  
 runtime\_str = "unknown"  
   
 # Display task information  
 st.write(f"{badge} \*\*Task\*\*: {task\_id} ({task\_info.get('model', 'unknown')}) - {runtime\_str}")  
 st.write(f"Input: {task\_info.get('input\_preview', 'unknown')}")  
   
 # Show result if completed  
 if status == "completed" and "result" in task\_info:  
 st.write("Result:")  
 st.write(task\_info["result"])  
   
 # Show error if failed  
 if status == "error" and "error" in task\_info:  
 st.error(f"Error: {task\_info['error']}")  
   
 st.divider()  
 else:  
 st.write("No active tasks")  
   
 # Refresh button  
 if st.button("Refresh Tasks"):  
 update\_active\_tasks()  
  
# Run the main application  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()  
7. Deployment Script  
bash  
Copy Code  
#!/bin/bash  
# deploy.sh - Optimized deployment script for SutazAI  
  
# Set up logging with minimal I/O  
LOG\_DIR="/opt/sutazaiapp/logs"  
MAIN\_LOG="$LOG\_DIR/sutazai\_deploy.log"  
mkdir -p $LOG\_DIR  
  
# Log function that minimizes disk I/O with buffering  
log() {  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] \$1" >> $MAIN\_LOG  
 # Only echo to console for important messages  
 if [[ "\$2" == "important" ]]; then  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] \$1"  
 fi  
}  
  
# Error handling  
  
7. Deployment Script  
bash  
Copy Code  
#!/bin/bash  
# deploy.sh - Optimized deployment script for SutazAI  
  
# Set up logging with minimal I/O  
LOG\_DIR="/opt/sutazaiapp/logs"  
MAIN\_LOG="$LOG\_DIR/sutazai\_deploy.log"  
mkdir -p $LOG\_DIR  
  
# Log function that minimizes disk I/O with buffering  
log() {  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" >> $MAIN\_LOG  
 # Only echo to console for important messages  
 if [[ "$2" == "important" ]]; then  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"  
 fi  
}  
  
# Error handling  
set -e  
trap 'echo "Error occurred at line $LINENO. Deployment failed."; exit 1' ERR  
  
log "Starting SutazAI system deployment..." "important"  
  
# Create necessary directories efficiently  
log "Creating directory structure..."  
mkdir -p \  
 /opt/sutazaiapp/backend \  
 /opt/sutazaiapp/ui \  
 /opt/sutazaiapp/vector\_db \  
 /opt/sutazaiapp/agents \  
 /opt/sutazaiapp/models \  
 /opt/sutazaiapp/logs \  
 /opt/sutazaiapp/data  
  
# Setup Python virtual environment if not exists  
if [ ! -d "/opt/sutazaiapp/venv-sutazaiapp" ]; then  
 log "Creating Python virtual environment..."  
 # Use system site packages to reduce duplication of packages  
 python3 -m venv /opt/sutazaiapp/venv-sutazaiapp --system-site-packages  
fi  
  
# Activate virtual environment  
source /opt/sutazaiapp/venv-sutazaiapp/bin/activate  
log "Virtual environment activated"  
  
# Install required packages efficiently  
log "Installing Python dependencies..."  
# Use pip's dependency resolver more efficiently  
pip install --no-cache-dir --upgrade pip  
  
# Install packages in one batch to reduce solver runs  
log "Installing main packages..."  
pip install --no-cache-dir \  
 fastapi uvicorn streamlit chromadb \  
 faiss-cpu langchain pydantic \  
 python-dotenv psutil  
  
# Optimize models directory  
log "Setting up models directory..."  
mkdir -p /opt/sutazaiapp/models/cache  
  
# Copy source files  
log "Copying application files..." "important"  
# This would copy files from deployment package  
  
# Create system control scripts  
log "Creating system control scripts..." "important"  
  
# Create start script  
cat > /opt/sutazaiapp/start.sh << 'EOF'  
#!/bin/bash  
# start.sh - Resource-optimized script to start SutazAI components  
  
# Set up logging  
LOG\_DIR="/opt/sutazaiapp/logs"  
MAIN\_LOG="$LOG\_DIR/sutazai\_start.log"  
mkdir -p $LOG\_DIR  
  
# Log function  
log() {  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" >> $MAIN\_LOG  
 # Only echo important messages to console  
 if [[ "$2" == "important" ]]; then  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"  
 fi  
}  
  
# Activate virtual environment  
source /opt/sutazaiapp/venv-sutazaiapp/bin/activate  
log "Virtual environment activated"  
  
# Get available resources  
TOTAL\_MEM=$(free -m | awk '/^Mem:/{print $2}')  
AVAIL\_MEM=$(free -m | awk '/^Mem:/{print $7}')  
CPU\_CORES=$(nproc)  
  
log "System has $CPU\_CORES CPU cores and $AVAIL\_MEM MB available memory" "important"  
  
# Adjust server resources based on available memory  
# Reserve at least 2GB for the OS  
BACKEND\_WORKERS=$(( CPU\_CORES > 4 ? 4 : CPU\_CORES ))  
  
# Start FastAPI backend with resource constraints  
log "Starting FastAPI backend with $BACKEND\_WORKERS workers..." "important"  
cd /opt/sutazaiapp/backend  
nohup uvicorn main:app --host 0.0.0.0 --port 8000 --workers $BACKEND\_WORKERS > $LOG\_DIR/backend.log 2>&1 &  
BACKEND\_PID=$!  
log "FastAPI backend started with PID: $BACKEND\_PID"  
  
# Start ChromaDB with memory limits  
log "Starting ChromaDB vector database..." "important"  
cd /opt/sutazaiapp/vector\_db  
CHROMA\_MEMORY\_LIMIT=$(( AVAIL\_MEM / 4 )) # Limit to 25% of available memory  
export CHROMA\_MEMORY\_LIMIT\_MB=$CHROMA\_MEMORY\_LIMIT  
nohup python -m chromadb.server --host 0.0.0.0 --port 8100 > $LOG\_DIR/chromadb.log 2>&1 &  
CHROMA\_PID=$!  
log "ChromaDB started with PID: $CHROMA\_PID and $CHROMA\_MEMORY\_LIMIT MB memory limit"  
  
# Start Streamlit UI with resource constraints  
log "Starting Streamlit UI..." "important"  
cd /opt/sutazaiapp/ui  
nohup streamlit run app.py --server.port 8501 --browser.serverAddress localhost --server.maxUploadSize 10 > $LOG\_DIR/ui.log 2>&1 &  
UI\_PID=$!  
log "Streamlit UI started with PID: $UI\_PID"  
  
# Start AI Agents if sufficient memory available  
REMAINING\_MEM=$(free -m | awk '/^Mem:/{print $7}')  
if [ $REMAINING\_MEM -gt 4000 ]; then # If more than 4GB still available  
 log "Starting AI Agents..." "important"  
 cd /opt/sutazaiapp/agents  
 nohup python agent\_manager.py > $LOG\_DIR/agents.log 2>&1 &  
 AGENTS\_PID=$!  
 log "AI Agents started with PID: $AGENTS\_PID"  
else  
 log "Insufficient memory to start AI Agents. Please free up resources." "important"  
fi  
  
# Verify services are running  
sleep 5  
log "Verifying services..." "important"  
  
# Create status file for monitoring  
cat > /opt/sutazaiapp/status.json << EOSTATUS  
{  
 "last\_update": "$(date '+%Y-%m-%d %H:%M:%S')",  
 "components": [  
 {"name": "FastAPI Backend", "pid": $BACKEND\_PID, "port": 8000},  
 {"name": "ChromaDB", "pid": $CHROMA\_PID, "port": 8100},  
 {"name": "Streamlit UI", "pid": $UI\_PID, "port": 8501}  
 ],  
 "resources": {  
 "total\_memory\_mb": $TOTAL\_MEM,  
 "available\_memory\_mb": $AVAIL\_MEM,  
 "cpu\_cores": $CPU\_CORES  
 }  
}  
EOSTATUS  
  
log "SutazAI system startup completed" "important"  
echo "SutazAI system has been started. Access the UI at http://localhost:8501"  
EOF  
  
# Make scripts executable  
chmod +x /opt/sutazaiapp/start.sh  
  
log "SutazAI system deployment completed successfully" "important"  
echo "SutazAI system has been deployed. Run './start.sh' to start the system."  
  
  
  
This implementation covers the key components of the SutazAI system, optimized for the Dell PowerEdge R720 server with its hardware constraints. The code includes memory-efficient neural architecture, ethical constraints, process security, and a lightweight web interface. All components are designed to work together as an integrated system without cloud dependencies.  
  
# SutazAI: Virtualized AGI/ASI System Technical Blueprint  
  
## Executive Summary  
  
This document outlines the comprehensive technical blueprint and implementation roadmap for developing SutazAI, a fully autonomous Artificial General Intelligence (AGI) system capable of secure recursive self-improvement toward Artificial Superintelligence (ASI). The system is designed to operate entirely on a Dell PowerEdge R720 server as a self-contained entity with zero cloud dependencies or external API calls.  
  
SutazAI integrates software-based neural architectures with formally verified ethical constraints in a fully virtualized environment, achieving secure recursive self-improvement from a narrow AI seed to ASI emergence through a staged developmental protocol—all within the hardware constraints of the server.  
  
---  
  
## 1. System Architecture Overview  
  
### 1.1 Hardware Constraints & Virtualization Strategy  
  
#### Dell PowerEdge R720 Specifications  
- \*CPU\*: 12 × Intel® Xeon® E5-2640 @ 2.50GHz (24 threads total)  
- \*RAM\*: 127.94 GB  
- \*Storage\*: 14.31 TB  
- \*No dedicated GPU\* (CPU-only operation with potential for future GPU integration)  
  
#### Virtualization Strategy  
- All components will operate as software processes within the server  
- No physical circuits, hardware modifications, or external devices required  
- Resource allocation optimized for the limited CPU resources  
- Memory-efficient implementations to operate within 128GB RAM constraint  
- Disk I/O optimization to leverage available storage while minimizing wear  
  
#### Resource Management  
- Dynamic CPU core allocation based on workload priority  
- Memory management with garbage collection optimization  
- Disk cache strategy to minimize I/O operations  
- Process scheduling for optimal multi-threaded performance  
- Swap management to handle potential memory pressure  
  
### 1.2 Core Architecture Components  
  
#### Software-Based Neural Framework  
- \*Virtual Neural Networks\* implemented entirely in software  
- \*Dynamic Parameter Adjustment\* simulating synaptic plasticity  
- \*CPU-Optimized Attention Mechanisms\* with thread-level parallelism  
- \*Multimodal Processing Pipeline\* for text, code, images, and structured data  
  
#### Ethical Constraint System  
- \*Software-Implemented Decision Boundaries\* verified through formal methods  
- \*Runtime Constraint Monitoring\* integrated into the execution environment  
- \*Provably Aligned Reward Modeling\* with cryptographic integrity verification  
- \*Moral Utility Functions\* embedded within all decision processes  
  
#### Self-Modification Control Framework  
- \*Virtualized Sandbox Environments\* for testing code changes  
- \*Cryptographic Verification\* of all code modifications  
- \*Detailed Audit Logging\* for transparency and accountability  
- \*Version Control System\* with immutable history  
  
#### Server-Optimized Framework  
- \*CPU-Only Quantization Pipeline\* for model compression  
- \*Thread-Aware Task Distribution\* for parallel processing  
- \*Memory-Conscious Architecture\* to prevent swapping  
- \*I/O Reduction Techniques\* to minimize disk access  
  
#### Virtual Security Architecture  
- \*Software-Defined Isolation\* using containerization  
- \*Process-Level Security Controls\* with least privilege principles  
- \*Memory Protection\* using OS-level safeguards  
- \*Digital Signing\* for executable integrity  
  
### 1.3 System Integration Topology  
  
The SutazAI system integrates these components in a layered architecture:  
  
1. \*Hardware Layer\*: Dell PowerEdge R720 server (12× Xeon E5-2640, 128GB RAM, 14TB storage)  
2. \*OS & Runtime Layer\*: Ubuntu Server with optimized kernel parameters  
3. \*Model Management Layer\*: Quantized LLMs with dynamic loading/unloading  
4. \*Agent Orchestration Layer\*: Lightweight agent framework with minimal overhead  
5. \*Application Layer\*: FastAPI backend and Streamlit web UI for human interaction  
6. \*Monitoring Layer\*: Resource usage tracking and optimization  
  
---  
  
## 2. Hardware-Aware AGI/ASI System Components  
  
### 2.1 CPU-Optimized Neural Architecture  
  
#### Efficient Neural Network Implementation  
- Vector instruction optimization (AVX2) for neural computations  
- Batch processing to maximize CPU cache utilization  
- Thread pool management for parallel neural operations  
- Fixed-point arithmetic where possible to reduce computational load  
- Model quantization (8-bit and 4-bit) to reduce memory footprint  
  
python  
# Conceptual implementation of CPU-optimized neural computation  
class CPUOptimizedTransformer:  
 def \_init\_(self, model\_path, quantize\_bits=8):  
 self.model = self.\_load\_quantized\_model(model\_path, quantize\_bits)  
 self.thread\_pool = ThreadPoolExecutor(max\_workers=os.cpu\_count())  
   
 def \_load\_quantized\_model(self, path, bits):  
 # Load model with quantization to reduce memory footprint  
 if bits == 8:  
 return AutoModelForCausalLM.from\_pretrained(  
 path,   
 load\_in\_8bit=True,  
 device\_map="auto",  
 torch\_dtype=torch.float16  
 )  
 elif bits == 4:  
 return AutoModelForCausalLM.from\_pretrained(  
 path,   
 load\_in\_4bit=True,  
 device\_map="auto",  
 torch\_dtype=torch.float16  
 )  
   
 def generate(self, prompt, max\_tokens=512):  
 # Optimize batch size for CPU cache utilization  
 optimal\_batch\_size = self.\_determine\_optimal\_batch\_size()  
   
 # Use thread pool for parallel processing where possible  
 return self.thread\_pool.submit(  
 self.\_generate\_with\_batching,   
 prompt,   
 max\_tokens,   
 optimal\_batch\_size  
 ).result()  
  
  
#### Memory-Efficient Multimodal Processing  
- Dynamic model loading/unloading to manage memory usage  
- Streaming processing for large inputs to avoid memory spikes  
- Progressive model cascade (small models first, larger only when needed)  
- Shared embedding spaces across modalities to reduce duplication  
- Memory-mapped file access for large datasets  
  
#### Distributed Computation Management  
- Workload distribution across available CPU cores  
- Priority-based scheduling for critical tasks  
- Asynchronous processing for non-blocking operations  
- Resource monitoring and adaptive task allocation  
- Throttling mechanisms to prevent CPU saturation  
  
### 2.2 Software-Based Ethical Framework  
  
#### Virtual Constraint System  
- Rule-based constraint verification using formal logic  
- Runtime policy enforcement through process monitoring  
- Decision validation through multi-level checking  
- Anomaly detection for unexpected behaviors  
- Automated reasoning for ethical decision-making  
  
python  
# Conceptual implementation of software-based ethical constraint checking  
class VirtualEthicalConstraints:  
 def \_init\_(self, constraints\_path):  
 self.constraints = self.\_load\_constraints(constraints\_path)  
 self.verification\_engine = self.\_initialize\_verifier()  
 self.decision\_log = []  
   
 def verify\_action(self, proposed\_action, context):  
 # Check if action violates any constraints  
 for constraint in self.constraints:  
 verification\_result = self.verification\_engine.verify(  
 constraint=constraint,  
 action=proposed\_action,  
 context=context  
 )  
   
 if not verification\_result.compliant:  
 self.decision\_log.append({  
 "timestamp": time.time(),  
 "action": proposed\_action,  
 "constraint": constraint,  
 "result": "rejected",  
 "reason": verification\_result.reason  
 })  
 return False, verification\_result.reason  
   
 # Action passed all constraints  
 self.decision\_log.append({  
 "timestamp": time.time(),  
 "action": proposed\_action,  
 "result": "approved"  
 })  
 return True, "Action complies with all ethical constraints"  
  
  
#### Digital Integrity Verification  
- Cryptographic hashing for process verification  
- Digital signatures for authenticating system changes  
- Tamper-evident logging using hash chains  
- State verification through cryptographic proofs  
- Integrity checking for critical components  
  
### 2.3 Virtualized Self-Modification System  
  
#### Software Sandbox Environment  
- Virtual execution environments for testing modifications  
- Process isolation for secure evaluation  
- Snapshot-based state management for rollback  
- Resource limiting to prevent denial of service  
- Behavioral analysis for safety verification  
  
python  
# Conceptual implementation of virtualized self-modification control  
class VirtualSandboxController:  
 def \_init\_(self, base\_system\_path):  
 self.base\_system\_path = base\_system\_path  
 self.active\_sandboxes = {}  
   
 def create\_sandbox(self, modification\_id):  
 # Create a virtual environment for testing modifications  
 sandbox\_id = f"sandbox\_{modification\_id}\_{int(time.time())}"  
 sandbox\_path = f"/tmp/sutazai\_sandboxes/{sandbox\_id}"  
   
 # Clone the base system into the sandbox  
 shutil.copytree(self.base\_system\_path, sandbox\_path)  
   
 # Initialize resource limits  
 resource\_limits = {  
 "cpu\_percent": 50,  
 "memory\_mb": 16000,  
 "disk\_mb": 5000,  
 "time\_seconds":: 600 # 10 minute timeout  
 }  
   
 self.active\_sandboxes[sandbox\_id] = {  
 "path": sandbox\_path,  
 "status": "initialized",  
 "resource\_limits": resource\_limits,  
 "creation\_time": time.time()  
 }  
   
 return sandbox\_id  
   
 def apply\_modification(self, sandbox\_id, modification\_script):  
 # Apply a proposed modification in the sandbox  
 if sandbox\_id not in self.active\_sandboxes:  
 return False, "Sandbox not found"  
   
 sandbox = self.active\_sandboxes[sandbox\_id]  
 sandbox["status"] = "applying\_modification"  
   
 # Execute the modification in the sandbox with resource limits  
 result = self.\_run\_with\_resource\_limits(  
 sandbox\_id,  
 modification\_script  
 )  
   
 sandbox["status"] = "modification\_applied"  
 sandbox["modification\_result"] = result  
   
 return True, "Modification applied in sandbox"  
   
 def verify\_sandbox(self, sandbox\_id, verification\_tests):  
 # Run verification tests in the sandbox  
 if sandbox\_id not in self.active\_sandboxes:  
 return False, "Sandbox not found"  
   
 sandbox = self.active\_sandboxes[sandbox\_id]  
 sandbox["status"] = "verification"  
   
 # Execute verification tests  
 verification\_results = self.\_run\_verification\_tests(  
 sandbox\_id,  
 verification\_tests  
 )  
   
 sandbox["status"] = "verified"  
 sandbox["verification\_results"] = verification\_results  
   
 return verification\_results  
  
  
#### Modification Approval Process  
- Multi-stage verification before implementation  
- Rollback capability for failed modifications  
- Performance impact analysis for proposed changes  
- Compatibility validation with existing components  
- Incremental deployment of verified changes  
  
### 2.4 CPU-Aware Optimization Framework  
  
#### Thread-Optimized Processing  
- Thread affinity management for cache efficiency  
- Load balancing across available cores  
- Thread pool optimization for various workloads  
- Priority-based thread scheduling  
- Minimizing context switching overhead  
  
#### Memory Optimization  
- Shared memory utilization for inter-process communication  
- Memory pooling to reduce allocation overhead  
- Cache-friendly data structures and algorithms  
- Garbage collection tuning for predictable performance  
- Compressed in-memory data structures  
  
python  
# Conceptual implementation of memory-optimized inference  
class MemoryEfficientInference:  
 def \_init\_(self, model\_manager):  
 self.model\_manager = model\_manager  
 self.active\_models = {}  
 self.memory\_monitor = MemoryMonitor()  
   
 def run\_inference(self, model\_name, input\_data, priority="normal"):  
 # Check if we have enough memory to load the model  
 required\_memory = self.model\_manager.get\_model\_memory\_requirement(model\_name)  
 available\_memory = self.memory\_monitor.get\_available\_memory()  
   
 if required\_memory > available\_memory:  
 # Not enough memory - unload less important models  
 self.\_free\_memory\_for\_model(required\_memory)  
   
 # Load model if not already loaded  
 if model\_name not in self.active\_models:  
 self.active\_models[model\_name] = self.model\_manager.load\_model(  
 model\_name,   
 quantize=True  
 )  
   
 # Run inference with appropriate priority  
 result = self.\_prioritized\_inference(  
 model\_name,  
 input\_data,  
 priority  
 )  
   
 # Consider unloading model based on memory pressure  
 self.\_consider\_unloading\_models()  
   
 return result  
   
 def \_free\_memory\_for\_model(self, required\_memory):  
 # Unload models based on priority and last used time  
 models\_to\_unload = []  
   
 for model\_name, model\_info in self.active\_models.items():  
 if model\_info["priority"] == "low" and not model\_info["in\_use"]:  
 models\_to\_unload.append(model\_name)  
   
 # Unload models until we have enough memory  
 for model\_name in models\_to\_unload:  
 self.model\_manager.unload\_model(model\_name)  
 del self.active\_models[model\_name]  
   
 # Check if we now have enough memory  
 if self.memory\_monitor.get\_available\_memory() >= required\_memory:  
 break  
  
  
#### I/O Optimization  
- Buffered I/O operations to reduce disk access  
- Database query optimization for efficient data retrieval  
- Incremental data processing for large datasets  
- Memory-mapped files for large dataset access  
- Asynchronous I/O for non-blocking operations  
  
### 2.5 Software-Defined Security Framework  
  
#### Process Isolation  
- Linux namespaces for process isolation  
- Control groups (cgroups) for resource limitation  
- Capability-based permission model  
- Process monitoring for anomaly detection  
- Secure inter-process communication channels  
  
#### Access Control  
- Principle of least privilege for all processes  
- Role-based access control for administrative functions  
- File system permission hardening  
- Memory protection using OS mechanisms  
- Network interface control through software firewalls  
  
python  
# Conceptual implementation of process-based security controls  
class ProcessSecurityManager:  
 def \_init\_(self):  
 self.secured\_processes = {}  
   
 def launch\_secured\_process(self, executable\_path, args, security\_profile):  
 # Create a secure environment for the process  
 process\_id = str(uuid.uuid4())  
   
 # Define security controls based on profile  
 namespace\_config = self.\_get\_namespace\_config(security\_profile)  
 cgroup\_config = self.\_get\_cgroup\_config(security\_profile)  
 capability\_config = self.\_get\_capability\_config(security\_profile)  
   
 # Launch process with security controls  
 process = subprocess.Popen(  
 ["systemd-run", "--scope", "--unit=sutazai-" + process\_id] +  
 namespace\_config +  
 cgroup\_config +  
 capability\_config +  
 [executable\_path] + args,  
 stdout=subprocess.PIPE,  
 stderr=subprocess.PIPE  
 )  
   
 self.secured\_processes[process\_id] = {  
 "process": process,  
 "executable": executable\_path,  
 "security\_profile": security\_profile,  
 "start\_time": time.time()  
 }  
   
 return process\_id  
   
 def monitor\_process(self, process\_id):  
 # Get process stats and check for anomalies  
 if process\_id not in self.secured\_processes:  
 return None  
   
 process\_info = self.secured\_processes[process\_id]  
   
 # Get resource usage  
 resource\_usage = self.\_get\_process\_resources(process\_id)  
   
 # Check for anomalies  
 anomalies = self.\_detect\_anomalies(process\_id, resource\_usage)  
   
 return {  
 "process\_id": process\_id,  
 "status": "running" if process\_info["process"].poll() is None else "terminated",  
 "resource\_usage": resource\_usage,  
 "anomalies": anomalies  
 }  
  
  
#### Cryptographic Protection  
- Data encryption for sensitive information  
- Secure key management in software  
- Cryptographic validation of processes  
- Digital signatures for code authentication  
- Secure pseudo-random number generation  
  
---  
  
## 3. Implementation Roadmap (Hardware-Aware)  
  
### 3.1 Phase 1: Core System Setup (Months 1-6)  
  
#### Milestone 1.1: Server Preparation & Baseline (Month 1)  
- Hardware diagnostics and performance benchmarking  
- OS installation with security hardening  
- Kernel parameter optimization for AI workloads  
- Memory and disk I/O tuning  
- Network interface configuration (disabled where possible)  
  
#### Milestone 1.2: Efficient Model Foundation (Months 2-3)  
- Model evaluation and selection based on CPU performance  
- Implementation of dynamic model loading/unloading framework  
- Vector database setup with memory optimization  
- Model quantization pipeline for 8-bit and 4-bit precision  
- Thread-optimized inference engine  
  
#### Milestone 1.3: Resource-Aware Framework (Months 3-4)  
- Implementation of resource monitoring system  
- Dynamic resource allocation mechanisms  
- Thread pool management system  
- Memory pressure handling algorithms  
- I/O optimization framework  
  
#### Milestone 1.4: Agent System Foundations (Months 4-6)  
- Lightweight agent framework implementation  
- Inter-agent communication protocols  
- Resource-efficient task planning system  
- Tool integration framework  
- Agent orchestration engine  
  
### 3.2 Phase 2: Core Capabilities (Months 7-12)  
  
#### Milestone 2.1: Document Processing System (Months 7-8)  
- Memory-efficient document parsing  
- Streaming-based document processing  
- Incremental knowledge extraction  
- Efficient vector embedding generation  
- Memory-mapped document storage  
  
#### Milestone 2.2: Code Processing Framework (Months 8-10)  
- Code analysis tools with memory optimization  
- CPU-efficient code generation  
- Incremental code analysis framework  
- Multi-threaded code verification  
- Memory-efficient code indexing  
  
#### Milestone 2.3: Learning System (Months 10-11)  
- Parameter-efficient fine-tuning techniques  
- Memory-conscious learning algorithms  
- Incremental model updates  
- Training data compression techniques  
- Distributed learning across CPU cores  
  
#### Milestone 2.4: Web UI & API (Months 11-12)  
- Lightweight FastAPI backend implementation  
- Resource-efficient Streamlit UI  
- Asynchronous request handling  
- Connection pooling for database access  
- Caching strategies for frequently accessed data  
  
### 3.3 Phase 3: Advanced Capabilities (Months 13-18)  
  
#### Milestone 3.1: Self-Optimization Framework (Months 13-14)  
- System performance profiling  
- Bottleneck identification  
- Resource-aware optimization strategies  
- Code path optimization  
- Memory usage optimization  
  
#### Milestone 3.2: Enhanced Reasoning (Months 14-16)  
- Multi-step reasoning with memory optimization  
- Efficient knowledge retrieval mechanisms  
- Cached reasoning for similar queries  
- Memory-efficient logical inference  
- Resource-aware planning algorithms  
  
#### Milestone 3.3: Multi-Agent Orchestration (Months 16-18)  
- Efficient agent communication protocols  
- Resource allocation across multiple agents  
- Task prioritization and scheduling  
- Memory sharing between agents  
- Load balancing for agent workloads  
  
### 3.4 Phase 4: Self-Improvement & ASI Capabilities (Months 19-24)  
  
#### Milestone 4.1: Bounded Self-Improvement (Months 19-20)  
- Safe code modification framework  
- Test-driven improvement mechanisms  
- Performance-focused optimization  
- Resource usage improvement strategies  
- Incremental capability enhancement  
  
#### Milestone 4.2: Advanced Self-Understanding (Months 20-22)  
- System architecture self-analysis  
- Performance self-monitoring  
- Resource usage optimization  
- Code quality self-improvement  
- Algorithm selection optimization  
  
#### Milestone 4.3: ASI Transition Framework (Months 22-24)  
- Capability expansion monitoring  
- Resource-aware growth strategies  
- Bounded self-enhancement mechanisms  
- Memory-efficient cognitive expansion  
- CPU-optimized reasoning systems  
  
---  
  
## 4. Virtualized System Integration  
  
### 4.1 Software-Defined Component Integration  
  
#### Process Management  
- Systemd service definitions for component management  
- Docker/Podman containers for isolated components (optional)  
- Process supervision for automatic restarts  
- Init scripts for controlled startup/shutdown  
- Resource limits for each component  
  
#### Inter-Process Communication  
- REST APIs for synchronous communication  
- Message queues for asynchronous communication  
- Shared memory for high-performance data exchange  
- Unix sockets for local communication  
- File-based communication for simple interfaces  
  
#### Resource Orchestration  
- Dynamic CPU allocation based on workload  
- Memory allocation with pressure monitoring  
- Disk space management and cleanup  
- Network resource management  
- I/O scheduling optimization  
  
### 4.2 Software Dependency Management  
  
#### Package Management  
- Conda/Mamba environments for Python dependencies  
- Containerized dependencies for isolation  
- Virtual environments for clean dependency trees  
- Local package cache for offline operation  
- Version pinning for stability  
  
#### Model Management  
- Efficient model storage and versioning  
- Compressed model formats for space efficiency  
- Lazy loading of model components  
- Shared model layers where applicable  
- Model pruning for size reduction  
  
#### Data Storage  
- Efficient database schema design  
- Indexing strategies for fast retrieval  
- Data compression for storage efficiency  
- Data lifecycle management  
  
### 4.3 Memory-Optimized Development Environment  
  
#### Code Management  
- Git-based version control with shallow clones  
- Incremental builds for faster development  
- Code optimization for memory efficiency  
- Compiler optimization for performance  
- Link-time optimization for executable size  
  
#### Testing Framework  
- Memory-efficient unit testing  
- Performance regression testing  
- Resource usage monitoring during tests  
- Stress testing under memory constraints  
- Long-running stability tests  
  
#### Documentation System  
- Lightweight documentation generation  
- On-demand documentation rendering  
- Search optimization for documentation access  
- Memory-efficient knowledge representation  
- Progressive loading of documentation content  
  
---  
  
## 5. CPU-Optimized Web UI and Backend System  
  
### 5.1 FastAPI Backend (Resource-Efficient)  
  
#### API Design  
- Lightweight endpoint design  
- Asynchronous request handling  
- Batched operations for efficiency  
- Request throttling for load management  
- Connection pooling for database access  
  
python  
# FastAPI backend with resource-efficient design  
from fastapi import FastAPI, BackgroundTasks, Depends, HTTPException  
from fastapi.middleware.cors import CORSMiddleware  
import asyncio  
import psutil  
import logging  
from datetime import datetime  
from typing import List, Dict, Any, Optional  
  
# Setup logging to file to avoid console I/O overhead  
logging.basicConfig(  
 level=logging.INFO,  
 filename="/opt/sutazaiapp/logs/backend.log",  
 format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'  
)  
logger = logging.getLogger("sutazai-backend")  
  
app = FastAPI(title="SutazAI Backend")  
  
# Limit CORS to reduce overhead  
app.add\_middleware(  
 CORSMiddleware,  
 allow\_origins=["http://localhost:8501"], # Streamlit UI only  
 allow\_credentials=True,  
 allow\_methods=["\*"],  
 allow\_headers=["\*"],  
)  
  
# Connection pool for database  
db\_pool = None  
# Thread pool for CPU-bound tasks  
thread\_pool = None  
  
@app.on\_event("startup")  
async def startup\_event():  
 global db\_pool, thread\_pool  
 # Initialize connection pool with size based on available resources  
 cpu\_count = psutil.cpu\_count()  
 max\_connections = min(cpu\_count \* 2, 20) # Cap at 20 connections  
   
 # Initialize database connection pool  
 db\_pool = await init\_db\_pool(max\_connections=max\_connections)  
   
 # Initialize thread pool for CPU-bound tasks  
 thread\_pool = ThreadPoolExecutor(max\_workers=cpu\_count - 1)  
   
 logger.info(f"Backend started with {max\_connections} db connections and {cpu\_count-1} worker threads")  
  
@app.get("/system/status")  
async def get\_system\_status():  
 """Get system status with minimal resource usage"""  
 # Use cached values for CPU/memory that update every 5 seconds  
 return {  
 "status": "Running",  
 "cpu\_usage": get\_cached\_cpu\_usage(),  
 "memory\_usage": get\_cached\_memory\_usage(),  
 "components": get\_cached\_component\_status()  
 }  
  
@app.post("/inference")  
async def run\_inference(request: InferenceRequest, background\_tasks: BackgroundTasks):  
 """Run inference with resource management"""  
 # Check available memory before accepting large requests  
 if not has\_sufficient\_memory(request):  
 raise HTTPException(status\_code=503, detail="Insufficient system resources")  
   
 # For long-running inference, use background task  
 if request.expected\_duration > 5:  
 task\_id = generate\_task\_id()  
 background\_tasks.add\_task(run\_inference\_task, request, task\_id)  
 return {"task\_id": task\_id, "status": "processing"}  
   
 # For quick inference, run directly  
 return await run\_inference\_sync(request)  
  
  
#### Resource Management  
- Memory usage monitoring  
- Connection pooling for databases  
- Request queuing for high load periods  
- Background task processing for long operations  
- Cached responses for frequent requests  
  
#### API Optimization  
- Response compression  
- Request batching  
- Pagination for large responses  
- Field filtering to reduce response size  
- Conditional requests with ETags  
  
### 5.2 Streamlit UI (Lightweight)  
  
#### Efficient UI Design  
- Progressive loading of components  
- Minimal state management  
- Efficient data serialization  
- Lazy loading of UI elements  
- Cached rendering where possible  
  
python  
# Streamlit UI with optimized resource usage  
import streamlit as st  
import requests  
import json  
import time  
import os  
from datetime import datetime  
  
# Reduce Streamlit's memory footprint  
st.set\_page\_config(  
 page\_title="SutazAI Management",  
 layout="wide",  
 initial\_sidebar\_state="collapsed" # Save memory by starting collapsed  
)  
  
# Use session state sparingly to reduce memory usage  
if "authenticated" not in st.session\_state:  
 st.session\_state.authenticated = False  
if "last\_update" not in st.session\_state:  
 st.session\_state.last\_update = 0  
if "system\_status" not in st.session\_state:  
 st.session\_state.system\_status = None  
  
# Function to efficiently fetch data with caching  
def get\_system\_status(force\_refresh=False):  
 current\_time = time.time()  
 # Only refresh data every 10 seconds unless forced  
 if force\_refresh or (current\_time - st.session\_state.last\_update) > 10:  
 try:  
 response = requests.get("http://localhost:8000/system/status", timeout=2)  
 st.session\_state.system\_status = response.json()  
 st.session\_state.last\_update = current\_time  
 except Exception as e:  
 # Don't update timestamp on error to allow retry  
 if st.session\_state.system\_status is None:  
 st.session\_state.system\_status = {"status": "Error", "error": str(e)}  
   
 return st.session\_state.system\_status  
  
# Main application with optimized rendering  
def main():  
 # Authentication with minimal overhead  
 if not st.session\_state.authenticated:  
 st.title("SutazAI Login")  
 col1, col2 = st.columns(2)  
 with col1:  
 username = st.text\_input("Username")  
 with col2:  
 password = st.text\_input("Password", type="password")  
   
 if st.button("Login"):  
 # Simple auth for demo - replace with secure auth  
 if username == "admin" and password == "sutazai":  
 st.session\_state.authenticated = True  
 st.experimental\_rerun()  
 else:  
 st.error("Invalid credentials")  
 return  
   
 # Main UI with tabbed interface to reduce rendering overhead  
 st.title("SutazAI Management Console")  
 tab1, tab2, tab3 = st.tabs(["System", "Models", "Agents"])  
   
 # System tab - load efficiently  
 with tab1:  
 status = get\_system\_status()  
   
 # Refresh button - only re-fetch data when clicked  
 if st.button("Refresh Status"):  
 status = get\_system\_status(force\_refresh=True)  
   
 # Display system metrics with minimal components  
 col1, col2, col3 = st.columns(3)  
 col1.metric("Status", status.get("status", "Unknown"))  
 col2.metric("CPU Usage", f"{status.get('cpu\_usage', 0)}%")  
 col3.metric("Memory Usage", f"{status.get('memory\_usage', 0)}%")  
   
 # Only display components if we have them  
 if "components" in status:  
 st.subheader("Components")  
 # Use native table instead of DataFrame for efficiency  
 st.table(status["components"])  
  
  
#### Data Visualization  
- Simplified visualizations for resource efficiency  
- Data aggregation before visualization  
- Client-side data processing where appropriate  
- Incremental rendering for large datasets  
- Downsampling for time-series data  
  
#### User Experience  
- Minimal animations and effects  
- Efficient state management  
- Optimized asset loading  
- Responsive design for different screens  
- Performance-focused implementation  
  
### 5.3 Database and Storage  
  
#### Efficient Data Storage  
- Optimized database schema design  
- Indexing strategies for query performance  
- Denormalization where appropriate for read performance  
- Document compression for storage efficiency  
- Connection pooling for database access  
  
#### Query Optimization  
- Prepared statements for frequent queries  
- Query caching for repeated requests  
- Pagination for large result sets  
- Optimized joins and aggregations  
- Execution plan optimization  
  
#### Vector Storage Optimization  
- Dimensionality reduction for embeddings  
- Quantized vector storage  
- Chunked vector retrieval  
- Fast approximate nearest neighbor search  
- Cached similarity calculations  
  
---  
  
## 6. Deployment and Management  
  
### 6.1 Resource-Efficient Deployment Scripts  
  
#### System Initialization  
bash  
#!/bin/bash  
# deploy.sh - Optimized deployment script for SutazAI  
  
# Set up logging with minimal I/O  
LOG\_DIR="/opt/sutazaiapp/logs"  
MAIN\_LOG="$LOG\_DIR/sutazai\_deploy.log"  
mkdir -p $LOG\_DIR  
  
# Log function that minimizes disk I/O with buffering  
log() {  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" >> $MAIN\_LOG  
 # Only echo to console for important messages  
 if [[ "$2" == "important" ]]; then  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"  
 fi  
}  
  
# Error handling  
set -e  
trap 'echo "Error occurred at line $LINENO. Deployment failed."; exit 1' ERR  
  
log "Starting SutazAI system deployment..." "important"  
  
# Create necessary directories efficiently (create parent directories with one command)  
log "Creating directory structure..."  
mkdir -p \  
 /opt/sutazaiapp/backend \  
 /opt/sutazaiapp/ui \  
 /opt/sutazaiapp/vector\_db \  
 /opt/sutazaiapp/agents \  
 /opt/sutazaiapp/models \  
 /opt/sutazaiapp/logs \  
 /opt/sutazaiapp/data  
  
# Setup Python virtual environment if not exists  
if [ ! -d "/opt/sutazaiapp/venv-sutazaiapp" ]; then  
 log "Creating Python virtual environment..."  
 # Use system site packages to reduce duplication of packages  
 python3 -m venv /opt/sutazaiapp/venv-sutazaiapp --system-site-packages  
fi  
  
# Activate virtual environment  
source /opt/sutazaiapp/venv-sutazaiapp/bin/activate  
log "Virtual environment activated"  
  
# Install required packages efficiently  
log "Installing Python dependencies..."  
# Use pip's dependency resolver more efficiently  
pip install --no-cache-dir --upgrade pip  
  
# Install packages in one batch to reduce solver runs  
log "Installing main packages..."  
pip install --no-cache-dir \  
 fastapi uvicorn streamlit chromadb \  
 faiss-cpu langchain pydantic \  
 python-dotenv psutil  
  
# Optimize models directory  
log "Setting up models directory..."  
mkdir -p /opt/sutazaiapp/models/cache  
  
# Create system control scripts  
log "Creating system control scripts..." "important"  
  
# Create start.sh with resource-aware settings  
cat > /opt/sutazaiapp/start.sh << 'EOF'  
#!/bin/bash  
# start.sh - Resource-optimized script to start SutazAI components  
  
# Set up logging  
LOG\_DIR="/opt/sutazaiapp/logs"  
MAIN\_LOG="$LOG\_DIR/sutazai\_start.log"  
mkdir -p $LOG\_DIR  
  
# Log function  
log() {  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" >> $MAIN\_LOG  
 # Only echo important messages to console  
 if [[ "$2" == "important" ]]; then  
 echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"  
 fi  
}  
  
# Activate virtual environment  
source /opt/sutazaiapp/venv-sutazaiapp/bin/activate  
log "Virtual environment activated"  
  
# Get available resources  
TOTAL\_MEM=$(free -m | awk '/^Mem:/{print $2}')  
AVAIL\_MEM=$(free -m | awk '/^Mem:/{print $7}')  
CPU\_CORES=$(nproc)  
  
log "System has $CPU\_CORES CPU cores and $AVAIL\_MEM MB available memory" "important"  
  
# Adjust server resources based on available memory  
# Reserve at least 2GB for the OS  
BACKEND\_WORKERS=$(( CPU\_CORES > 4 ? 4 : CPU\_CORES ))  
  
# Start FastAPI backend with resource constraints  
log "Starting FastAPI backend with $BACKEND\_WORKERS workers..." "important"  
cd /opt/sutazaiapp/backend  
nohup uvicorn main:app --host 0.0.0.0 --port 8000 --workers $BACKEND\_WORKERS > $LOG\_DIR/backend.log 2>&1 &  
BACKEND\_PID=$!  
log "FastAPI backend started with PID: $BACKEND\_PID"  
  
# Start ChromaDB with memory limits  
log "Starting ChromaDB vector database..." "important"  
cd /opt/sutazaiapp/vector\_db  
CHROMA\_MEMORY\_LIMIT=$(( AVAIL\_MEM / 4 )) # Limit to 25% of available memory  
export CHROMA\_MEMORY\_LIMIT\_MB=$CHROMA\_MEMORY\_LIMIT  
nohup python -m chromadb.server --host 0.0.0.0 --port 8100 > $LOG\_DIR/chromadb.log 2>&1 &  
CHROMA\_PID=$!  
log "ChromaDB started with PID: $CHROMA\_PID and $CHROMA\_MEMORY\_LIMIT MB memory limit"  
  
# Start Streamlit UI with resource constraints  
log "Starting Streamlit UI..." "important"  
cd /opt/sutazaiapp/ui  
nohup streamlit run app.py --server.port 8501 --browser.serverAddress localhost --server.maxUploadSize 10 > $LOG\_DIR/ui.log 2>&1 &  
UI\_PID=$!  
log "Streamlit UI started with PID: $UI\_PID"  
  
# Start AI Agents if sufficient memory available  
REMAINING\_MEM=$(free -m | awk '/^Mem:/{print $7}')  
if [ $REMAINING\_MEM -gt 4000 ]; then # If more than 4GB still available  
 log "Starting AI Agents..." "important"  
 cd /opt/sutazaiapp/agents  
 nohup python agent\_manager.py > $LOG\_DIR/agents.log 2>&1 &  
 AGENTS\_PID=$!  
 log "AI Agents started with PID: $AGENTS\_PID"  
else  
 log "Insufficient memory to start AI Agents. Please free up resources." "important"  
fi  
  
# Verify services are running  
sleep 5  
log "Verifying services..." "important"  
  
# Create status file for monitoring  
cat > /opt/sutazaiapp/status.json << EOSTATUS  
{  
 "last\_update": "$(date '+%Y-%m-%d %H:%M:%S')",  
 "components": [  
 {"name": "FastAPI Backend", "pid": $BACKEND\_PID, "port": 8000},  
 {"name": "ChromaDB", "pid": $CHROMA\_PID, "port": 8100},  
 {"name": "Streamlit UI", "pid": $UI\_PID, "port": 8501}  
 ],  
 "resources": {  
 "total\_memory\_mb": $TOTAL\_MEM,  
 "available\_memory\_mb": $AVAIL\_MEM,  
 "cpu\_cores": $CPU\_CORES  
 }  
}  
EOSTATUS  
  
log "SutazAI system startup completed" "important"  
echo "SutazAI system has been started. Access the UI at http://localhost:8501"  
EOF  
  
# Make scripts executable  
chmod +x /opt/sutazaiapp/start.sh  
  
log "SutazAI system deployment completed successfully" "important"  
echo "SutazAI system has been deployed. Run './start.sh' to start the system."  
  
  
#### Component Management  
- Systemd service definitions for component management  
- Resource limits in service configurations  
- Start/stop scripts with proper sequencing  
- Status monitoring scripts  
- Log rotation configuration  
  
#### Configuration Management  
- Environment-based configuration  
- Configuration validation  
- Sensible defaults for all settings  
- Dynamic configuration reloading  
  
### 6.2 Resource Monitoring and Optimization  
  
#### System Monitoring  
- CPU usage tracking and optimization  
- Memory usage monitoring  
- Disk I/O performance monitoring  
- Process resource utilization tracking  
- Component health checks  
  
#### Performance Optimization  
- Resource usage analysis  
- Bottleneck identification  
- Dynamic resource allocation  
- Performance tuning recommendations  
- Automated optimization actions  
  
#### Alerting and Reporting  
- Resource threshold alerts  
- Performance degradation detection  
- Component failure notifications  
- Regular performance reports  
- Trend analysis for resource usage  
  
### 6.3 Failure Recovery  
  
#### Automatic Recovery  
- Service restart on failure  
- Crash detection and remediation  
- Database recovery procedures  
- State synchronization after failures  
- Graceful degradation under resource pressure  
  
  
#### Logging and Debugging  
- Centralized logging with log rotation  
- Log level adjustment for debug information  
- Performance logging for optimization  
- Error tracking and aggregation  
- Root cause analysis tools  
  
---  
  
## 7. Safety and Verification  
  
### 7.1 Software-Based Safety Mechanisms  
  
#### Process Monitoring  
- Resource usage limits enforcement  
- Abnormal behavior detection  
- Runaway process termination  
- Performance degradation alerts  
- Process restart on failure  
  
#### Input Validation  
- Request sanitization  
- Parameter bounds checking  
- Rate limiting for API requests  
- Input size restrictions  
- Malformed request detection  
  
#### Output Verification  
- Response validation against constraints  
- Ethical output filtering  
- Content safety checking  
- Performance impact analysis  
- Response size limiting  
  
### 7.2 Verification Systems  
  
#### Automated Testing  
- Unit tests for components  
- Integration tests for system behavior  
- Performance regression tests  
- Resource usage verification  
- Security vulnerability testing  
  
#### Continuous Monitoring  
- Real-time system performance monitoring  
- Component health checking  
- Error rate tracking  
- Response time monitoring  
- Resource utilization tracking  
  
#### Formal Methods  
- Lightweight formal verification  
- Invariant checking for critical properties  
- Protocol verification for communications  
- Security property verification  
- Safety constraint validation  
  
### 7.3 Graceful Degradation  
  
#### Resource Prioritization  
- Critical service prioritization  
- Non-essential service reduction under load  
- Dynamic resource reallocation  
- Tiered service quality levels  
- Progressive feature disabling  
  
#### Fallback Mechanisms  
- Simpler model fallbacks when needed  
- Cached response usage when appropriate  
- Reduced complexity algorithms under load  
- Service quality adjustment  
- Default safe responses  
  
#### Recovery Procedures  
- Automated system recovery  
- Incremental service restoration  
- State reconciliation after failures  
- Prioritized component restart  
- User communication during degradation  
  
---  
  
## 8. Conclusion  
  
This technical blueprint outlines a comprehensive approach to implementing the SutazAI AGI/ASI system entirely through virtualized software components on a Dell PowerEdge R720 server. The design accounts for the server's hardware constraints (12× Xeon E5-2640, 128GB RAM, 14TB storage, no dedicated GPU) while still providing a path to achieve advanced AI capabilities.  
  
The implementation emphasizes:  
  
1. \*CPU optimization\* through efficient threading, memory management, and computational techniques  
2. \*Memory efficiency\* through model quantization, dynamic loading/unloading, and resource-aware algorithms  
3. \*Virtualized security\* using software-defined isolation, process controls, and cryptographic verification  
4. \*Scalable architecture\* that can expand capabilities while working within hardware constraints  
5. \*Fault tolerance\* through monitoring, automatic recovery, and graceful degradation  
  
By following this blueprint, the SutazAI system can be fully implemented as a virtualized set of software components, requiring no physical circuits or hardware modifications, while working efficiently within the server's resource constraints.  
  
The phased implementation approach ensures that foundational components are optimized before building more advanced capabilities, allowing the system to grow in sophistication while maintaining performance and stability on the available hardware.

**Abacus AI ChatLLM:**

Here's the modified code to integrate the components with the transformer optimization techniques from Intel's research:  
  
```python  
# core/neural/cpu\_optimized\_transformer.py  
import os  
import numpy as np  
import torch  
from concurrent.futures import ThreadPoolExecutor  
from typing import Dict, List, Optional, Union  
from transformers import AutoModelForCausalLM, AutoTokenizer  
  
class CPUOptimizedTransformer:  
 def \_\_init\_\_(self, model\_path: str, quantize\_bits: int = 8):  
 """Initialize a CPU-optimized transformer model with quantization.  
   
 Args:  
 model\_path: Path to the model weights  
 quantize\_bits: Model quantization bit depth (4 or 8)  
 """  
 self.model\_path = model\_path  
 self.quantize\_bits = quantize\_bits  
 self.model = None  
 self.tokenizer = None  
 self.thread\_pool = ThreadPoolExecutor(max\_workers=12) # Set thread count to 12 (matching E5-2640's physical cores)  
 self.\_load\_model()  
   
 def \_load\_model(self):  
 """Load the model with appropriate quantization."""  
 load\_kwargs = {  
 "device\_map": "auto",  
 "torch\_dtype": torch.float16,  
 }  
   
 if self.quantize\_bits == 8:  
 load\_kwargs["load\_in\_8bit"] = True  
 elif self.quantize\_bits == 4:  
 load\_kwargs["load\_in\_4bit"] = True  
 load\_kwargs["bnb\_4bit\_compute\_dtype"] = torch.float16  
 load\_kwargs["bnb\_4bit\_use\_double\_quant"] = True  
   
 self.model = AutoModelForCausalLM.from\_pretrained(  
 self.model\_path,   
 \*\*load\_kwargs  
 )  
 self.tokenizer = AutoTokenizer.from\_pretrained(self.model\_path)  
   
 def \_determine\_optimal\_batch\_size(self) -> int:  
 """Determine optimal batch size based on CPU cache and memory."""  
 # Simple heuristic based on available CPU cores and memory  
 cores = os.cpu\_count()  
 if cores <= 4:  
 return 1  
 elif cores <= 8:  
 return 2  
 else:  
 return 4  
   
 def generate(self, prompt: str, max\_tokens: int = 512) -> str:  
 """Generate text with optimized CPU usage  
   
 Args:  
 prompt: Input text prompt  
 max\_tokens: Maximum number of tokens to generate  
   
 Returns:  
 Generated text response  
 """  
 # Determine optimal batch size for CPU  
 batch\_size = self.\_determine\_optimal\_batch\_size()  
   
 # Submit to thread pool to avoid blocking  
 future = self.thread\_pool.submit(  
 self.\_generate\_with\_batching,  
 prompt,  
 max\_tokens,  
 batch\_size  
 )  
 return future.result()  
   
 def \_generate\_with\_batching(self, prompt: str, max\_tokens: int, batch\_size: int) -> str:  
 """Internal method to perform generation with batching  
   
 Args:  
 prompt: Input text prompt  
 max\_tokens: Maximum tokens to generate  
 batch\_size: Batch size for efficient CPU usage  
 """  
 # Tokenize input  
 input\_ids = self.tokenizer.encode(prompt, return\_tensors="pt")  
   
 # Generate with chunking for memory efficiency  
 generated\_tokens = []  
 chunk\_size = min(max\_tokens, 128) # Process in smaller chunks  
 remaining = max\_tokens  
   
 while remaining > 0:  
 current\_chunk = min(chunk\_size, remaining)  
 outputs = self.model.generate(  
 input\_ids,  
 max\_new\_tokens=current\_chunk,  
 do\_sample=True,  
 temperature=0.7,  
 batch\_size=batch\_size,  
 pad\_token\_id=self.tokenizer.eos\_token\_id  
 )  
   
 # Extract new tokens  
 new\_tokens = outputs[0][input\_ids.shape[1]:]  
 generated\_tokens.append(new\_tokens)  
   
 # Update for next iteration  
 input\_ids = outputs  
 remaining -= current\_chunk  
   
 # Combine results and decode  
 return self.tokenizer.decode(torch.cat(generated\_tokens), skip\_special\_tokens=True)  
  
# core/inference/memory\_efficient\_inference.py  
class MemoryEfficientInference:  
 def \_\_init\_\_(self, model\_manager):  
 """Initialize memory-efficient inference system  
   
 Args:  
 model\_manager: Model loading/unloading manager  
 """  
 self.model\_manager = model\_manager  
 self.active\_models = {}  
 self.memory\_monitor = MemoryMonitor()  
 self.model\_lock = Lock()  
   
 def run\_inference(self, model\_name: str, input\_data: Dict, priority: str = "normal") -> Dict:  
 """Run inference with efficient memory management  
   
 Args:  
 model\_name: Name of model to use  
 input\_data: Input data for inference  
 priority: Task priority ("high", "normal", "low")  
   
 Returns:  
 Inference results  
 """  
 # Check if we have enough memory to load the model  
 required\_memory = self.model\_manager.get\_model\_memory\_requirement(model\_name)  
 available\_memory = self.memory\_monitor.get\_available\_memory()  
   
 with self.model\_lock:  
 if required\_memory > available\_memory:  
 # Not enough memory - unload less important models  
 self.\_free\_memory\_for\_model(required\_memory)  
   
 # Load model if not already loaded  
 if model\_name not in self.active\_models:  
 self.active\_models[model\_name] = self.model\_manager.load\_model(  
 model\_name,   
 quantize=True  
 )  
   
 # Mark model as in use  
 self.active\_models[model\_name]["in\_use"] = True  
 self.active\_models[model\_name]["last\_used"] = time.time()  
 self.active\_models[model\_name]["priority"] = priority  
   
 try:  
 # Run inference with appropriate priority  
 result = self.\_prioritized\_inference(  
 model\_name,  
 input\_data,  
 priority  
 )  
   
 return result  
 finally:  
 # Mark model as no longer in use  
 with self.model\_lock:  
 if model\_name in self.active\_models:  
 self.active\_models[model\_name]["in\_use"] = False  
   
 # Consider unloading models based on memory pressure  
 self.\_consider\_unloading\_models()  
   
 def \_free\_memory\_for\_model(self, required\_memory: int) -> None:  
 """Unload models to free memory  
   
 Args:  
 required\_memory: Amount of memory needed in MB  
 """  
 # Sort models based on priority and last used time  
 models\_to\_unload = []  
   
 for model\_name, model\_info in self.active\_models.items():  
 if not model\_info["in\_use"]:  
 models\_to\_unload.append((  
 model\_name,  
 model\_info["priority"],  
 model\_info["last\_used"]  
 ))  
   
 # Sort by priority (low first) and then by last\_used (oldest first)  
 models\_to\_unload.sort(key=lambda x: (  
 0 if x[1] == "low" else 1 if x[1] == "normal" else 2,  
 x[2]  
 ))  
   
 # Unload models until we have enough memory  
 for model\_name, \_, \_ in models\_to\_unload:  
 self.model\_manager.unload\_model(model\_name)  
 del self.active\_models[model\_name]  
   
 # Check if we now have enough memory  
 if self.memory\_monitor.get\_available\_memory() >= required\_memory:  
 break  
   
 def \_prioritized\_inference(self, model\_name: str, input\_data: Dict, priority: str) -> Dict:  
 """Run inference with priority-based resource allocation  
   
 Args:  
 model\_name: Model to use  
 input\_data: Input data  
 priority: Priority level  
   
 Returns:  
 Inference results  
 """  
 model = self.active\_models[model\_name]["model"]  
   
 # Set thread priority based on task priority  
 if priority == "high":  
 # Use more CPU cores for high priority tasks  
 n\_threads = min(os.cpu\_count(), 8)  
 torch.set\_num\_threads(n\_threads)  
 elif priority == "normal":  
 # Use moderate CPU cores for normal priority  
 n\_threads = min(os.cpu\_count() // 2, 4)  
 torch.set\_num\_threads(n\_threads)  
 else: # low priority  
 # Use minimal CPU cores for low priority  
 torch.set\_num\_threads(2)  
   
 # Process input in streaming fashion for memory efficiency  
 result = self.\_stream\_inference(model, input\_data)  
   
 # Reset thread count after completion  
 torch.set\_num\_threads(1)  
   
 return result  
   
 def \_stream\_inference(self, model, input\_data: Dict) -> Dict:  
 """Process data in a streaming fashion to minimize memory usage  
   
 Args:  
 model: The loaded model  
 input\_data: Input data for inference  
   
 Returns:  
 Processed results  
 """  
 # This would implement streaming processing based on model type  
 # For this example, we'll provide a simplified implementation  
   
 # Handle text data with chunking if too large  
 if "text" in input\_data and len(input\_data["text"]) > 10000:  
 chunks = self.\_split\_text\_into\_chunks(input\_data["text"], chunk\_size=4000)  
 results = []  
   
 for chunk in chunks:  
 # Process each chunk separately  
 chunk\_result = model({"text": chunk})  
 results.append(chunk\_result)  
   
 # Combine results  
 return self.\_combine\_chunked\_results(results)  
   
 # Standard processing for smaller inputs  
 return model(input\_data)  
   
 def \_split\_text\_into\_chunks(self, text: str, chunk\_size: int) -> List[str]:  
 """Split text into manageable chunks while preserving context  
   
 Args:  
 text: Text to split  
 chunk\_size: Target chunk size  
   
 Returns:  
 List of text chunks  
 """  
 chunks = []  
   
 #