AAMAS System Architecture & Design

Autonomous AI Multi-Agent Software Development Platform

SYSTEM OVERVIEW

AAMAS is designed as a cloud-native, microservices-based platform that orchestrates specialized AI agents to automate software development workflows. The architecture emphasizes scalability, reliability, and intelligent agent coordination.

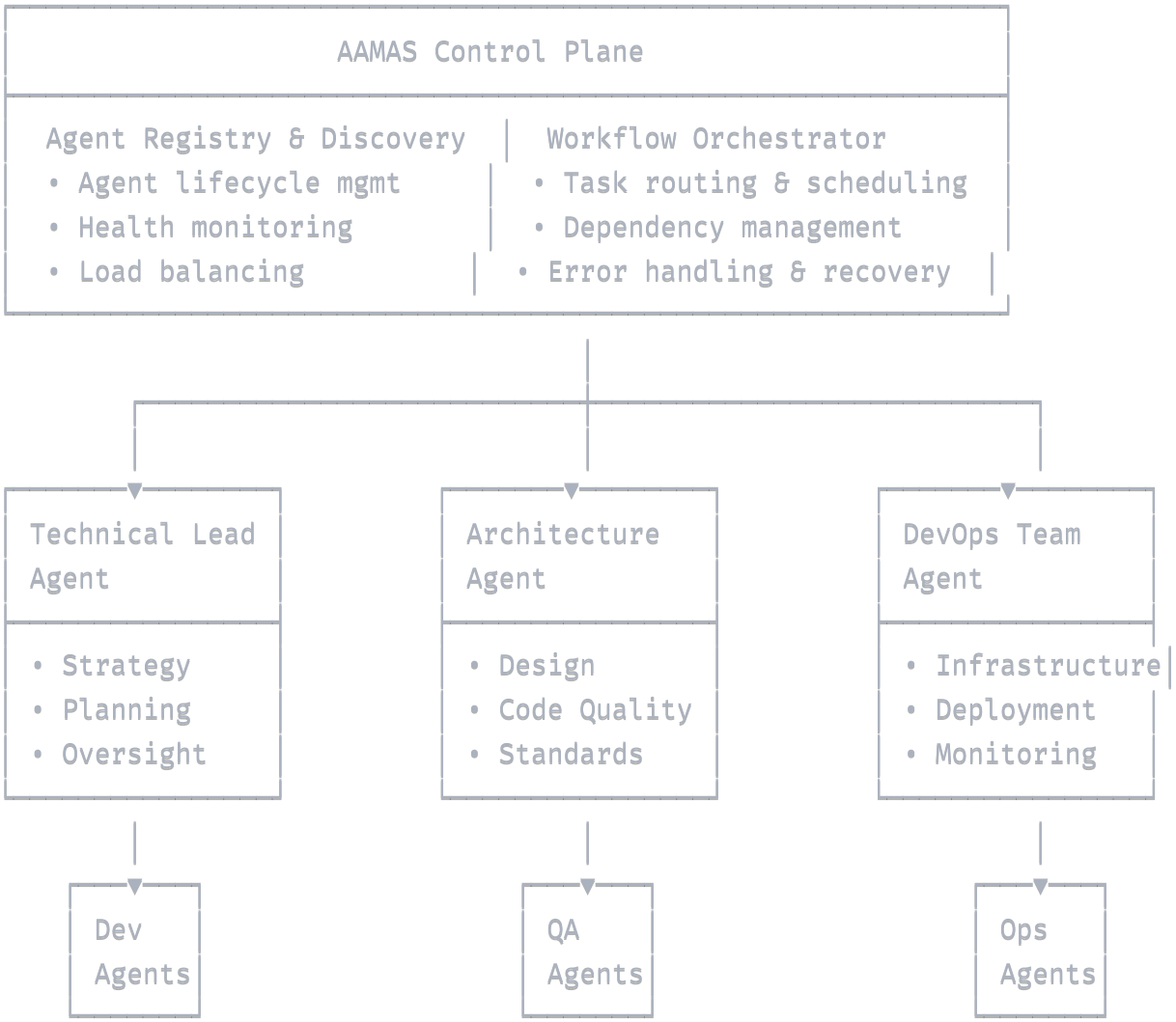
High-Level Architecture Principles

- **Agent-Centric Design**: Every major function is handled by specialized AI agents
- **Event-Driven Architecture**: Asynchronous communication between agents and services
- **Microservices Pattern**: Loosely coupled, independently deployable services
- **Multi-Tenant SaaS**: Secure isolation between customer environments
- **Cloud-Native**: Kubernetes-based deployment with auto-scaling capabilities

---

CORE SYSTEM ARCHITECTURE

1. Agent Orchestration Layer



2. Multi-Agent Communication System

python

```python
# Agent Communication Protocol
class AgentCommunicationBus:
    def __init__(self):
        self.message_broker = KafkaProducer()
        self.agent_registry = AgentRegistry()

    async def send_task(self, from_agent: str, to_agent: str, task: Task):
        """Route tasks between agents with priority and routing logic"""
        message = TaskMessage(
            sender=from_agent,
            recipient=to_agent,
            task=task,
            timestamp=datetime.utcnow(),
            correlation_id=uuid4()
        )

        # Route through appropriate channel based on task type
        topic = self.get_routing_topic(task.type)
        await self.message_broker.send(topic, message)

    async def broadcast_event(self, event: SystemEvent):
        """Broadcast system-wide events to all relevant agents"""
        interested_agents = self.agent_registry.get_subscribers(event.type)
        for agent_id in interested_agents:
            await self.send_event(agent_id, event)
```

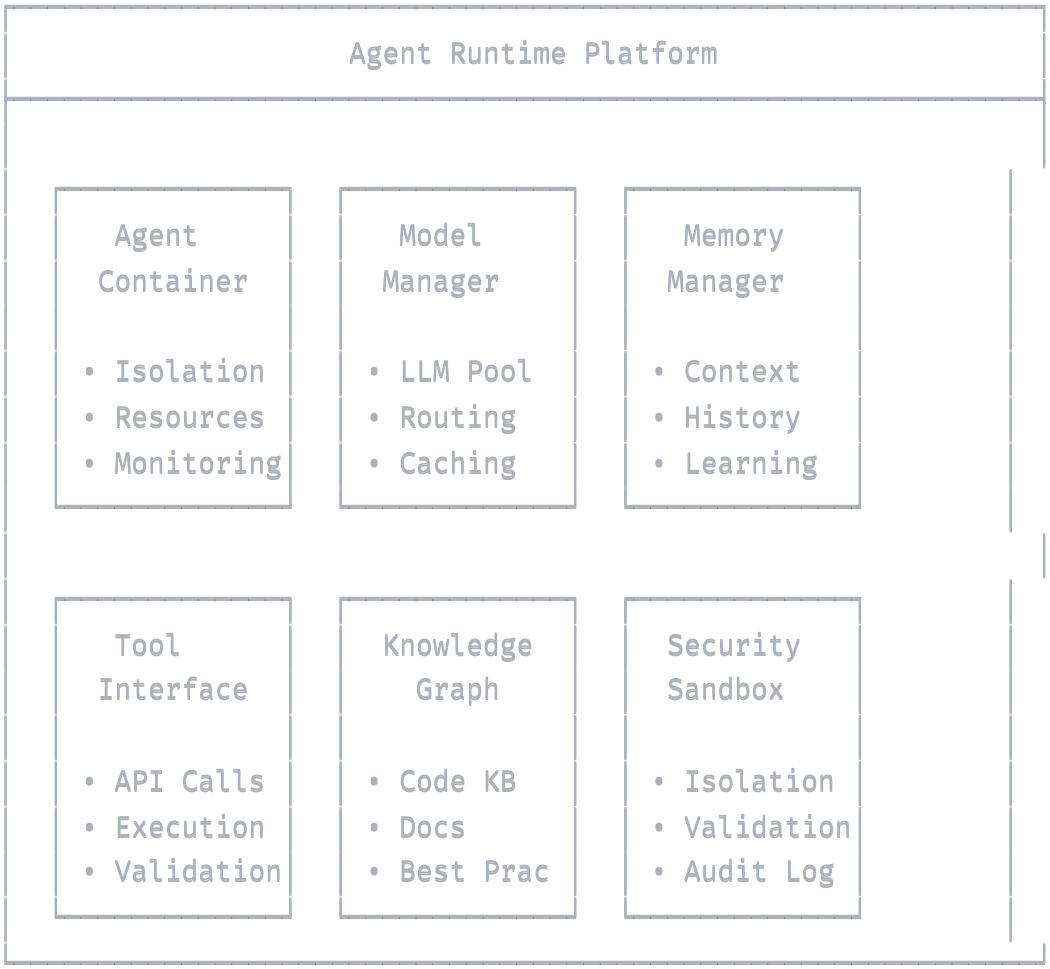
3. Agent State Management

python

```python
class AgentStateManager:
    """Manages persistent state for all agents across the system"""

    def __init__(self):
        self.state_store = RedisCluster()  # For fast access
        self.persistent_store = MongoDB()   # For long-term storage

    async def save_agent_state(self, agent_id: str, state: dict):
        """Save agent state with versioning and history"""
        state_record = AgentState(
            agent_id=agent_id,
            state=state,
            version=await self.get_next_version(agent_id),
            timestamp=datetime.utcnow()
        )

        # Fast cache for active state
        await self.state_store.set(f"agent:{agent_id}:current", state_record)

        # Persistent storage for history and recovery
        await self.persistent_store.insert_one(state_record.dict())
```

DETAILED COMPONENT ARCHITECTURE

1. AI Agent Runtime Environment



2. Specialized Agent Implementations

Technical Lead Agent

python

```python
class TechnicalLeadAgent(BaseAgent):
    """Strategic oversight and technical decision-making"""

    def __init__(self):
        super().__init__(
            role="technical_lead",
            capabilities=[
                "strategic_planning",
                "risk_assessment",
                "resource_allocation",
                "quality_oversight"
            ]
        )
        self.decision_engine = TechnicalDecisionEngine()
        self.risk_analyzer = RiskAnalyzer()

    async def evaluate_project_health(self, project_id: str):
        """Comprehensive project health assessment"""
        metrics = await self.get_project_metrics(project_id)
        risks = await self.risk_analyzer.analyze(project_id)
        team_performance = await self.assess_team_velocity(project_id)

        health_score = self.calculate_health_score(
            metrics, risks, team_performance
        )

        recommendations = await self.generate_recommendations(
            health_score, risks
        )

        return ProjectHealthReport(
            score=health_score,
            risks=risks,
            recommendations=recommendations,
            next_review=datetime.utcnow() + timedelta(days=7)
        )
```

Architecture Agent

```
python
```

```python
class ArchitectureAgent(BaseAgent):
    """System design and code quality optimization"""

    def __init__(self):
        super().__init__(
            role="architecture",
            capabilities=[
                "system_design",
                "code_review",
                "pattern_recognition",
                "refactoring_suggestions"
            ]
        )
        self.design_patterns = DesignPatternLibrary()
        self.quality_analyzer = CodeQualityAnalyzer()

    async def review_architecture_change(self, change_request: ArchitectureChange):
        """Analyze proposed architecture changes for impact and quality"""

        # Analyze current system state
        current_architecture = await self.analyze_current_system(
            change_request.project_id
        )

        # Simulate proposed changes
        impact_analysis = await self.simulate_change_impact(
            current_architecture, change_request
        )

        # Check against best practices
        compliance_check = await self.check_design_principles(
            change_request.proposed_design
        )

        # Generate recommendations
        recommendations = await self.generate_architecture_recommendations(
            impact_analysis, compliance_check
        )

        return ArchitectureReview(
            impact_score=impact_analysis.risk_level,
            compliance_score=compliance_check.score,
            recommendations=recommendations,
```

```
            approval_status=self.determine_approval_status(impact_analysis)
        )
```

3. Development Team Agents

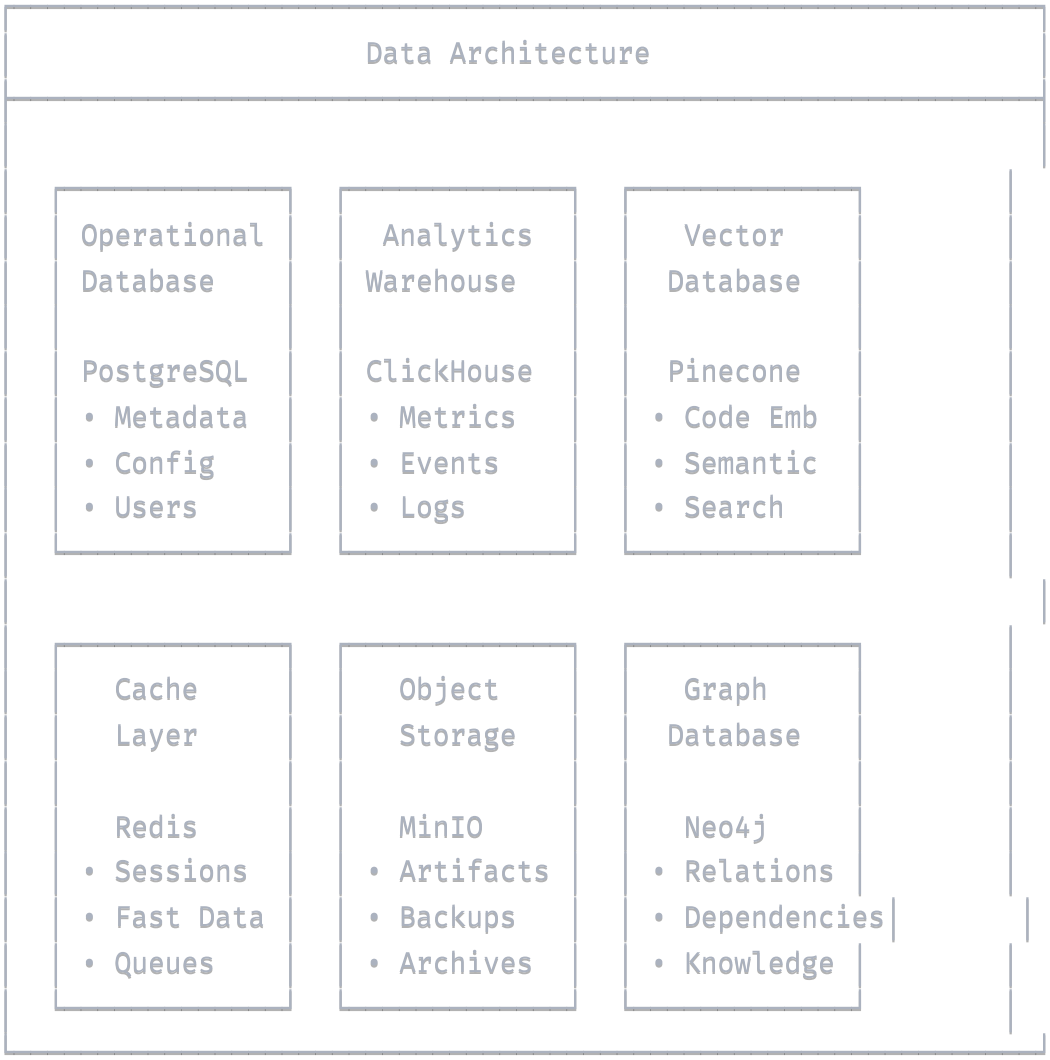
```
python
```

```python
class DevelopmentTeamAgent(BaseAgent):
    """Feature implementation and code generation"""

    def __init__(self, specialization: str):
        super().__init__(
            role=f"developer_{specialization}",
            capabilities=[
                "code_generation",
                "testing",
                "debugging",
                "documentation"
            ]
        )
        self.code_generator = CodeGenerator(specialization)
        self.test_generator = TestGenerator()

    async def implement_feature(self, feature_spec: FeatureSpecification):
        """Full feature implementation with tests and documentation"""

        # Generate implementation plan
        implementation_plan = await self.plan_implementation(feature_spec)

        # Generate code
        code_artifacts = await self.code_generator.generate_code(
            feature_spec, implementation_plan
        )

        # Generate tests
        test_suite = await self.test_generator.generate_tests(
            code_artifacts, feature_spec
        )

        # Generate documentation
        documentation = await self.generate_documentation(
            code_artifacts, feature_spec
        )

        # Self-review
        review_result = await self.self_review_code(code_artifacts)

        return FeatureImplementation(
            code=code_artifacts,
            tests=test_suite,
```

```
            documentation=documentation,
            quality_metrics=review_result.metrics,
            estimated_effort=implementation_plan.effort_estimate
        )
```

DATA ARCHITECTURE

1. Multi-Modal Data Storage Strategy



2. Data Models

```
python
```

```python
# Core Data Models
from dataclasses import dataclass
from typing import List, Dict, Optional
from enum import Enum

@dataclass
class Project:
    id: str
    name: str
    organization_id: str
    repository_url: str
    tech_stack: List[str]
    agents_assigned: List[str]
    configuration: Dict
    created_at: datetime
    updated_at: datetime

@dataclass
class Agent:
    id: str
    type: AgentType
    specialization: str
    model_version: str
    capabilities: List[str]
    current_projects: List[str]
    performance_metrics: Dict
    status: AgentStatus
    last_active: datetime

@dataclass
class Task:
    id: str
    type: TaskType
    priority: Priority
    assigned_agent: str
    project_id: str
    description: str
    requirements: Dict
    dependencies: List[str]
    status: TaskStatus
    created_at: datetime
    due_date: Optional[datetime]
```

```python
@dataclass
class CodeArtifact:
    id: str
    project_id: str
    file_path: str
    content: str
    language: str
    generated_by: str
    quality_score: float
    test_coverage: float
    documentation_score: float
    created_at: datetime
```

INTEGRATION ARCHITECTURE

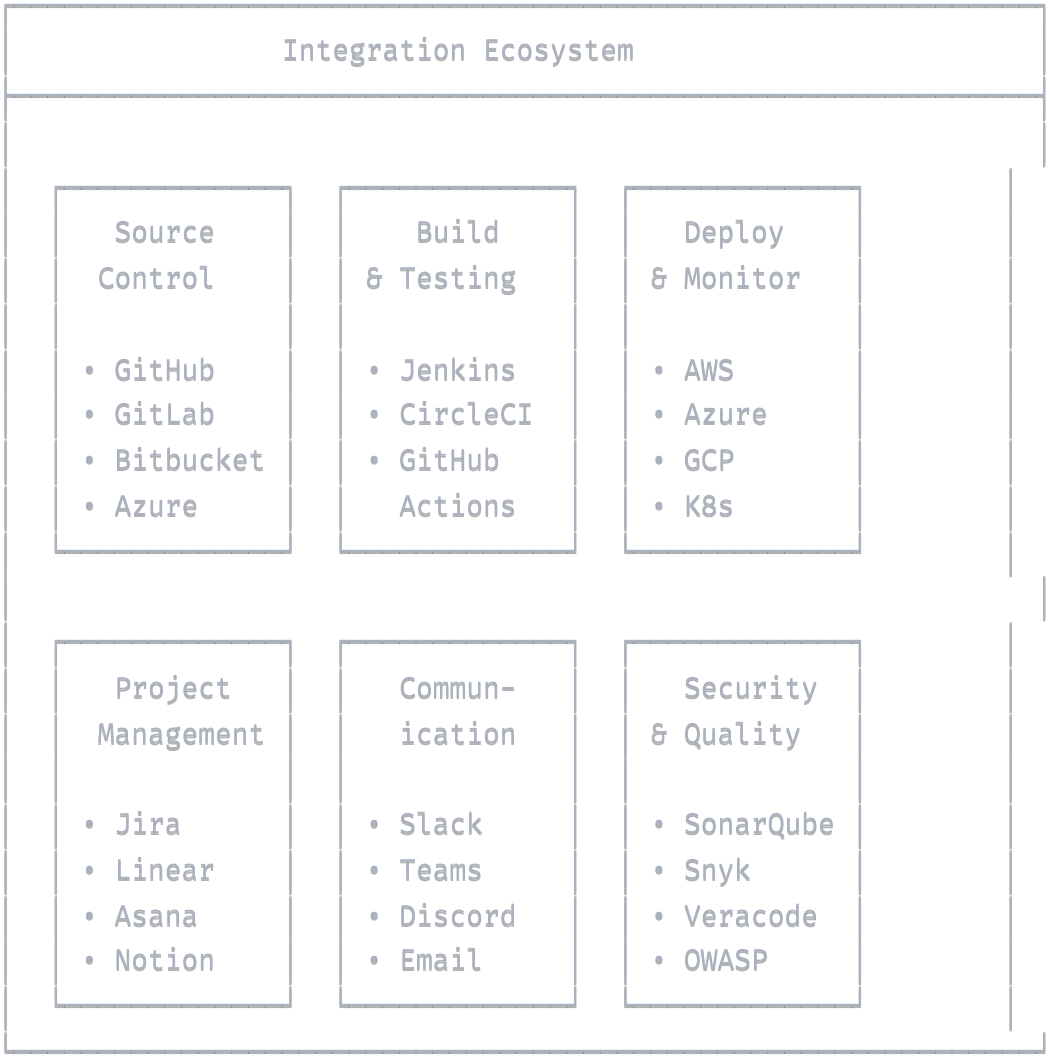
1. External Tool Integration Layer

```
python
```

```python
class IntegrationManager:
    """Manages all external tool integrations"""

    def __init__(self):
        self.integrations = {
            'git': GitIntegration(),
            'ci_cd': CICDIntegration(),
            'monitoring': MonitoringIntegration(),
            'communication': CommunicationIntegration()
        }

    async def register_webhook(self, tool: str, project_id: str, events: List[str]):
        """Register webhooks for external tool events"""
        integration = self.integrations[tool]
        webhook_url = f"{self.base_url}/webhooks/{tool}/{project_id}"

        await integration.register_webhook(webhook_url, events)

    async def handle_webhook(self, tool: str, project_id: str, payload: dict):
        """Process incoming webhook events"""
        event = self.parse_webhook_event(tool, payload)

        # Route to appropriate agents
        interested_agents = await self.get_interested_agents(event)

        for agent_id in interested_agents:
            await self.notify_agent(agent_id, event)

class GitIntegration:
    """Git repository integration"""

    async def setup_project_integration(self, project: Project):
        """Set up git hooks and monitoring for a project"""

        # Set up branch protection rules
        await self.setup_branch_protection(project.repository_url)

        # Configure automated PR creation
        await self.setup_pr_automation(project)

        # Set up commit analysis
        await self.setup_commit_hooks(project)
```

```python
async def create_pull_request(self, project_id: str, changes: CodeChanges):
    """Create PR with AI-generated description and reviews"""

    pr_description = await self.generate_pr_description(changes)
    reviewers = await self.suggest_reviewers(changes)

    pr = await self.git_client.create_pr(
        title=changes.title,
        description=pr_description,
        reviewers=reviewers,
        labels=changes.labels
    )

    return pr
```

2. Development Tool Ecosystem



SECURITY ARCHITECTURE

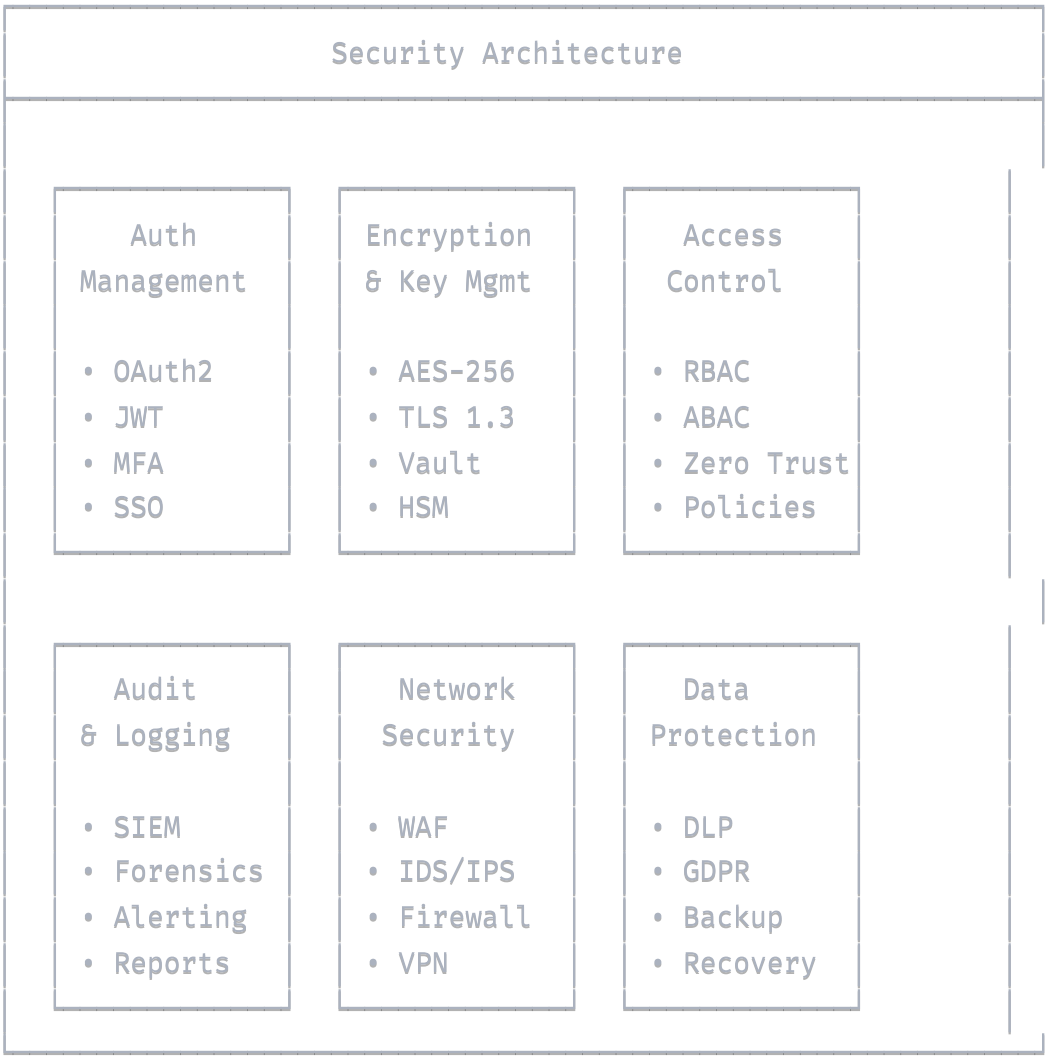
1. Multi-Layered Security Model

```
python
```

```python
class SecurityManager:
    """Comprehensive security management for the platform"""

    def __init__(self):
        self.auth_manager = AuthenticationManager()
        self.encryption_manager = EncryptionManager()
        self.audit_logger = AuditLogger()
        self.access_controller = AccessController()

    async def secure_agent_communication(self, message: AgentMessage):
        """Secure all inter-agent communications"""

        # Encrypt message content
        encrypted_content = await self.encryption_manager.encrypt(
            message.content, message.recipient
        )

        # Add digital signature
        signature = await self.create_message_signature(
            message.sender, encrypted_content
        )

        # Log communication for audit
        await self.audit_logger.log_agent_communication(
            sender=message.sender,
            recipient=message.recipient,
            message_type=message.type,
            timestamp=datetime.utcnow()
        )

        return SecureMessage(
            content=encrypted_content,
            signature=signature,
            metadata=message.metadata
        )

class CodeSecurityAnalyzer:
    """Analyze code for security vulnerabilities"""

    async def analyze_code_security(self, code: str, language: str):
        """Comprehensive security analysis of generated code"""

        findings = []
```

```
        # Static security analysis
        static_results = await self.static_security_scan(code, language)
        findings.extend(static_results)

        # Pattern-based vulnerability detection
        pattern_results = await self.pattern_vulnerability_scan(code)
        findings.extend(pattern_results)

        # Dependency security check
        dependency_results = await self.dependency_security_scan(code)
        findings.extend(dependency_results)

        return SecurityAnalysisResult(
            findings=findings,
            risk_level=self.calculate_risk_level(findings),
            recommendations=self.generate_security_recommendations(findings)
        )
```

2. Data Protection & Privacy



SCALABILITY & PERFORMANCE

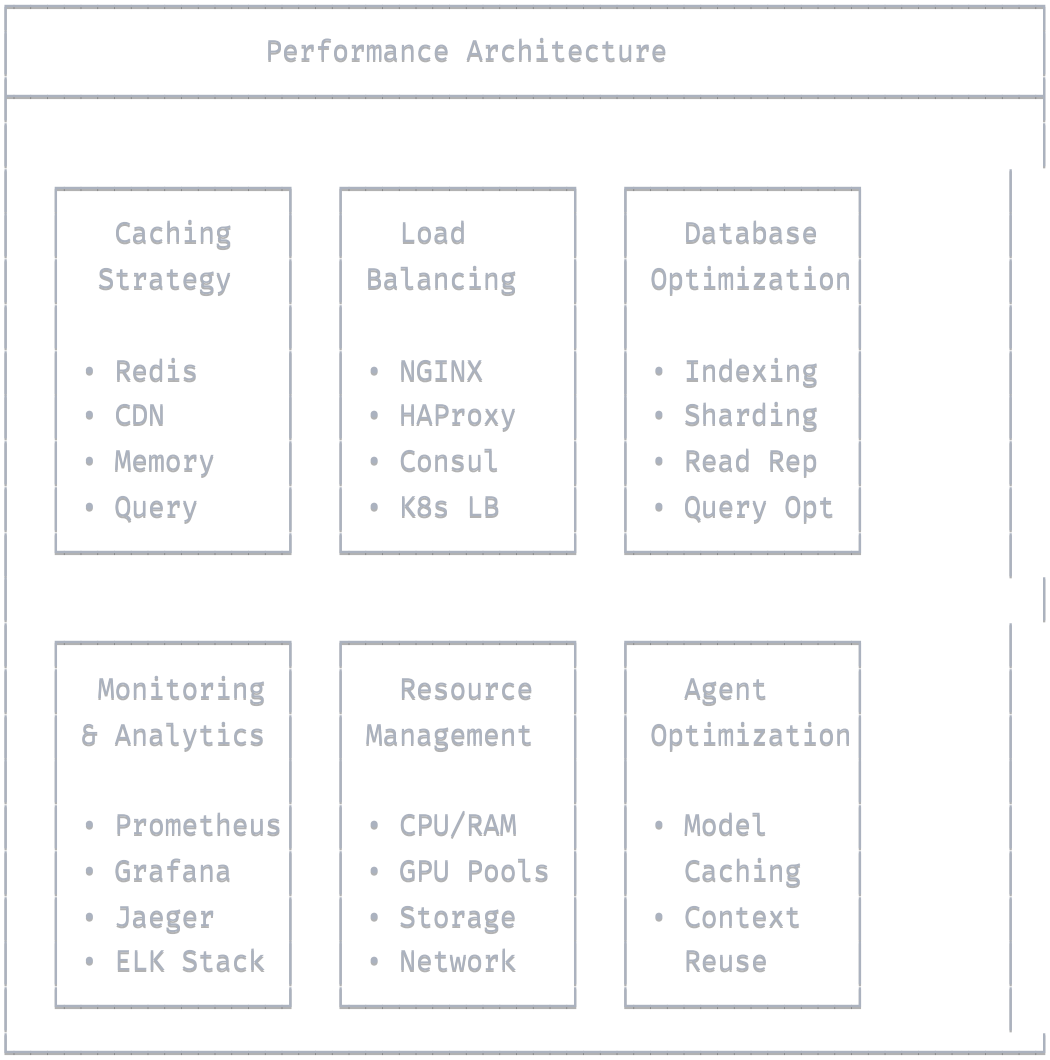
1. Auto-Scaling Architecture

```
python
```

```python
class ScalingManager:
    """Manages dynamic scaling of agents and infrastructure"""

    def __init__(self):
        self.metrics_collector = MetricsCollector()
        self.resource_manager = ResourceManager()
        self.agent_pool = AgentPool()

    async def monitor_and_scale(self):
        """Continuous monitoring and scaling decisions"""

        while True:
            # Collect system metrics
            metrics = await self.metrics_collector.get_current_metrics()

            # Analyze scaling needs
            scaling_decision = await self.analyze_scaling_needs(metrics)

            if scaling_decision.should_scale_up:
                await self.scale_up_agents(scaling_decision.agent_types)
            elif scaling_decision.should_scale_down:
                await self.scale_down_agents(scaling_decision.agent_types)

            # Infrastructure scaling
            if scaling_decision.needs_infrastructure_scaling:
                await self.scale_infrastructure(scaling_decision.infrastructure_needs

            await asyncio.sleep(30)  # Check every 30 seconds

    async def scale_up_agents(self, agent_types: List[str]):
        """Dynamically add agent instances"""

        for agent_type in agent_types:
            # Determine optimal scaling strategy
            scaling_config = await self.get_scaling_config(agent_type)

            # Create new agent instances
            new_agents = await self.agent_pool.create_agents(
                agent_type, scaling_config.instance_count
            )

            # Register and activate
            for agent in new_agents:
```

```
            await self.register_agent(agent)
            await self.activate_agent(agent)
```

2. Performance Optimization



DEPLOYMENT ARCHITECTURE

1. Container Orchestration

yaml

```yaml
# Kubernetes Deployment Configuration
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aamas-technical-lead-agent
spec:
  replicas: 3
  selector:
    matchLabels:
      app: technical-lead-agent
  template:
    metadata:
      labels:
        app: technical-lead-agent
    spec:
      containers:
      - name: technical-lead-agent
        image: aamas/technical-lead-agent:v1.0.0
        resources:
          requests:
            memory: "2Gi"
            cpu: "1000m"
            nvidia.com/gpu: 1
          limits:
            memory: "4Gi"
            cpu: "2000m"
            nvidia.com/gpu: 1
        env:
        - name: AGENT_TYPE
          value: "technical_lead"
        - name: MODEL_ENDPOINT
          valueFrom:
            secretKeyRef:
              name: model-config
              key: endpoint
        ports:
        - containerPort: 8080
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 10
```

```
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
```

2. CI/CD Pipeline

python

```python
# CI/CD Pipeline Configuration
class AAMASPipeline:
    """Automated build, test, and deployment pipeline"""

    stages = [
        "code_quality_check",
        "security_scan",
        "unit_tests",
        "integration_tests",
        "agent_performance_tests",
        "staging_deployment",
        "end_to_end_tests",
        "production_deployment",
        "monitoring_setup"
    ]

    async def execute_pipeline(self, commit: Commit):
        """Execute full CI/CD pipeline"""

        pipeline_run = PipelineRun(
            commit=commit,
            stages=self.stages,
            started_at=datetime.utcnow()
        )

        for stage in self.stages:
            stage_result = await self.execute_stage(stage, commit)
            pipeline_run.add_stage_result(stage_result)

            if not stage_result.success:
                await self.handle_pipeline_failure(pipeline_run, stage)
                break

        return pipeline_run
```

MONITORING & OBSERVABILITY

1. Comprehensive Monitoring Stack

```
python
```

```python
class MonitoringSystem:
    """Complete observability for the AAMAS platform"""

    def __init__(self):
        self.metrics_collector = PrometheusCollector()
        self.log_aggregator = ELKStack()
        self.tracing_system = Jaeger()
        self.alerting_system = AlertManager()

    async def setup_agent_monitoring(self, agent: Agent):
        """Set up comprehensive monitoring for an agent"""

        # Performance metrics
        await self.metrics_collector.register_agent_metrics(
            agent_id=agent.id,
            metrics=[
                "response_time",
                "task_completion_rate",
                "error_rate",
                "resource_utilization",
                "model_accuracy"
            ]
        )

        # Distributed tracing
        await self.tracing_system.setup_agent_tracing(agent.id)

        # Log aggregation
        await self.log_aggregator.configure_agent_logging(
            agent_id=agent.id,
            log_level="INFO"
        )

        # Alerts
        await self.setup_agent_alerts(agent)

    async def setup_agent_alerts(self, agent: Agent):
        """Configure alerting for agent health and performance"""

        alerts = [
            Alert(
                name=f"{agent.id}_high_response_time",
                condition="avg_response_time > 5s",
```

```
                severity="warning",
                actions=["scale_up", "notify_ops"]
            ),
            Alert(
                name=f"{agent.id}_error_rate",
                condition="error_rate > 5%",
                severity="critical",
                actions=["restart_agent", "notify_ops", "fallback_mode"]
            )
        ]

        for alert in alerts:
            await self.alerting_system.register_alert(alert)
```

2. Real-Time Dashboard & Analytics

```python
class AAMASDashboard:
    """Real-time monitoring dashboard for AAMAS platform"""

    def __init__(self):
        self.dashboard_server = GrafanaDashboard()
        self.real_time_updater = WebSocketManager()
        self.analytics_engine = AnalyticsEngine()

    async def create_customer_dashboard(self, customer_id: str):
        """Create customized dashboard for each customer"""

        dashboard_config = {
            "customer_id": customer_id,
            "panels": [
                {
                    "title": "Agent Performance Overview",
                    "type": "stat_panel",
                    "metrics": [
                        "active_agents",
                        "tasks_completed_today",
                        "average_response_time",
                        "success_rate"
                    ]
                },
                {
                    "title": "Development Velocity",
                    "type": "time_series",
```

```
                    "metrics": [
                        "commits_per_day",
                        "pull_requests_created",
                        "bugs_fixed",
                        "features_delivered"
                    ]
                },
                {
                    "title": "Code Quality Trends",
                    "type": "gauge",
                    "metrics": [
                        "code_quality_score",
                        "test_coverage",
                        "technical_debt_ratio",
                        "security_score"
                    ]
                },
                {
                    "title": "Resource Utilization",
                    "type": "heatmap",
                    "metrics": [
                        "cpu_usage",
                        "memory_usage",
                        "gpu_utilization",
                        "storage_usage"
                    ]
                }
            ]
        }

        dashboard = await self.dashboard_server.create_dashboard(dashboard_config)

        # Set up real-time updates
        await self.real_time_updater.setup_dashboard_updates(
            customer_id, dashboard.id
        )

        return dashboard

class AnalyticsEngine:
    """Advanced analytics for performance optimization"""

    async def generate_insights(self, customer_id: str, timeframe: str):
        """Generate actionable insights from platform usage"""
```

```python
        # Collect data
        usage_data = await self.collect_usage_data(customer_id, timeframe)
        performance_data = await self.collect_performance_data(customer_id, timefram
        quality_data = await self.collect_quality_data(customer_id, timeframe)

        # AI-powered analysis
        insights = await self.ai_analyzer.analyze_patterns(
            usage_data, performance_data, quality_data
        )

        # Generate recommendations
        recommendations = await self.generate_recommendations(insights)

        return AnalyticsReport(
            timeframe=timeframe,
            insights=insights,
            recommendations=recommendations,
            roi_metrics=self.calculate_roi_metrics(usage_data, performance_data)
        )
```

API ARCHITECTURE

1. RESTful API Design

```
python
```

```python
from fastapi import FastAPI, HTTPSRedirect, Depends
from fastapi.security import HTTPBearer
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI(
    title="AAMAS Platform API",
    description="Autonomous AI Multi-Agent Software Development Platform",
    version="1.0.0"
)

# Security middleware
security = HTTPBearer()
app.add_middleware(HTTPSRedirect)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://*.aamas.dev"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.post("/api/v1/projects", response_model=Project)
async def create_project(
    project_data: CreateProjectRequest,
    current_user: User = Depends(get_current_user)
):
    """Create a new project with AI agent assignment"""

    # Validate project configuration
    validation_result = await validate_project_config(project_data)
    if not validation_result.is_valid:
        raise HTTPException(400, validation_result.errors)

    # Create project
    project = await project_service.create_project(
        project_data, current_user.organization_id
    )

    # Auto-assign appropriate agents
    agent_assignments = await agent_manager.assign_agents_to_project(
        project.id, project_data.requirements
    )
```

```python
    # Initialize project environment
    await project_initializer.setup_project_environment(
        project, agent_assignments
    )

    return project

@app.get("/api/v1/projects/{project_id}/agents", response_model=List[AgentStatus])
async def get_project_agents(
    project_id: str,
    current_user: User = Depends(get_current_user)
):
    """Get all agents assigned to a project with their current status"""

    # Check permissions
    await check_project_access(project_id, current_user)

    # Get agent assignments
    agents = await agent_manager.get_project_agents(project_id)

    # Get current status for each agent
    agent_statuses = []
    for agent in agents:
        status = await agent_monitor.get_agent_status(agent.id)
        agent_statuses.append(status)

    return agent_statuses

@app.post("/api/v1/projects/{project_id}/tasks", response_model=Task)
async def create_task(
    project_id: str,
    task_data: CreateTaskRequest,
    current_user: User = Depends(get_current_user)
):
    """Create a new task and assign to appropriate agent"""

    await check_project_access(project_id, current_user)

    # Analyze task requirements
    task_analysis = await task_analyzer.analyze_task(task_data)

    # Find best agent for the task
    optimal_agent = await agent_matcher.find_optimal_agent(
        project_id, task_analysis
```

```
    )

    # Create and assign task
    task = await task_service.create_task(
        project_id=project_id,
        task_data=task_data,
        assigned_agent=optimal_agent.id,
        estimated_effort=task_analysis.effort_estimate
    )

    # Notify agent
    await agent_communicator.assign_task(optimal_agent.id, task)

    return task

@app.get("/api/v1/analytics/dashboard/{customer_id}")
async def get_dashboard_data(
    customer_id: str,
    timeframe: str = "7d",
    current_user: User = Depends(get_current_user)
):
    """Get dashboard analytics data for a customer"""

    await check_customer_access(customer_id, current_user)

    # Generate real-time analytics
    dashboard_data = await analytics_engine.generate_dashboard_data(
        customer_id, timeframe
    )

    return dashboard_data

# WebSocket endpoint for real-time updates
@app.websocket("/ws/project/{project_id}/updates")
async def project_updates_websocket(
    websocket: WebSocket,
    project_id: str,
    token: str = Query(...)
):
    """Real-time project updates via WebSocket"""

    # Authenticate WebSocket connection
    user = await authenticate_websocket_token(token)
    await check_project_access(project_id, user)
```

```
    await websocket.accept()

    # Subscribe to project events
    event_subscription = await event_bus.subscribe(
        f"project.{project_id}.*", websocket
    )

    try:
        while True:
            # Keep connection alive and handle incoming messages
            data = await websocket.receive_text()
            message = json.loads(data)

            if message["type"] == "ping":
                await websocket.send_json({"type": "pong"})
            elif message["type"] == "subscribe":
                await handle_subscription_request(message, websocket)

    except WebSocketDisconnect:
        await event_subscription.unsubscribe()
```

2. GraphQL API for Complex Queries

```
python
```

```python
import strawberry
from strawberry.fastapi import GraphQLRouter

@strawberry.type
class Project:
    id: str
    name: str
    status: str
    agents: List['Agent']
    tasks: List['Task']
    metrics: 'ProjectMetrics'

@strawberry.type
class Agent:
    id: str
    type: str
    specialization: str
    status: str
    current_tasks: List['Task']
    performance_metrics: 'AgentMetrics'

@strawberry.type
class Task:
    id: str
    title: str
    description: str
    status: str
    assigned_agent: Agent
    estimated_effort: float
    actual_effort: Optional[float]
    quality_score: Optional[float]

@strawberry.type
class Query:
    @strawberry.field
    async def project(self, id: str, info: Info) -> Optional[Project]:
        """Get detailed project information with nested data"""

        user = await get_current_user_from_context(info.context)
        await check_project_access(id, user)

        # Use DataLoader pattern for efficient data fetching
        project_loader = info.context["project_loader"]
```

```python
        project = await project_loader.load(id)

        return project

    @strawberry.field
    async def projects(
        self,
        filter: Optional[ProjectFilter] = None,
        pagination: Optional[PaginationInput] = None,
        info: Info = None
    ) -> ProjectConnection:
        """Get filtered and paginated list of projects"""

        user = await get_current_user_from_context(info.context)

        # Build query with filters
        query_builder = ProjectQueryBuilder()
        query = query_builder.build_query(filter, user.organization_id)

        # Execute with pagination
        results = await project_service.get_projects(
            query, pagination or PaginationInput()
        )

        return ProjectConnection(
            edges=[ProjectEdge(node=project) for project in results.projects],
            page_info=results.page_info,
            total_count=results.total_count
        )

@strawberry.type
class Mutation:
    @strawberry.mutation
    async def create_project(
        self,
        input: CreateProjectInput,
        info: Info
    ) -> CreateProjectPayload:
        """Create a new project with intelligent agent assignment"""

        user = await get_current_user_from_context(info.context)

        try:
            # Create project with auto-agent assignment
```

```
            project = await project_service.create_project_with_agents(
                input, user.organization_id
            )

            return CreateProjectPayload(
                project=project,
                success=True,
                errors=[]
            )
        except ValidationError as e:
            return CreateProjectPayload(
                project=None,
                success=False,
                errors=e.errors
            )

# Create GraphQL schema
schema = strawberry.Schema(query=Query, mutation=Mutation)
graphql_app = GraphQLRouter(schema)

# Add to FastAPI app
app.include_router(graphql_app, prefix="/graphql")
```

DISASTER RECOVERY & BACKUP

1. Multi-Region Backup Strategy

```
python
```

```python
class DisasterRecoveryManager:
    """Comprehensive disaster recovery and backup management"""

    def __init__(self):
        self.backup_scheduler = BackupScheduler()
        self.replication_manager = ReplicationManager()
        self.recovery_orchestrator = RecoveryOrchestrator()

    async def setup_multi_region_backup(self, customer_id: str):
        """Set up multi-region backup for customer data"""

        backup_config = BackupConfiguration(
            customer_id=customer_id,
            backup_frequency="hourly",
            retention_policy={
                "hourly": "24h",
                "daily": "30d",
                "weekly": "12w",
                "monthly": "12m"
            },
            regions=["us-east-1", "us-west-2", "eu-west-1"],
            encryption_enabled=True,
            compression_enabled=True
        )

        # Set up automated backups
        await self.backup_scheduler.schedule_backups(backup_config)

        # Configure cross-region replication
        await self.replication_manager.setup_replication(backup_config)

        # Test backup integrity
        await self.test_backup_integrity(customer_id)

    async def execute_disaster_recovery(self, incident: DisasterIncident):
        """Execute disaster recovery procedures"""

        recovery_plan = await self.generate_recovery_plan(incident)

        # Notify stakeholders
        await self.notify_incident_response_team(incident, recovery_plan)

        # Execute recovery steps
```

```
        for step in recovery_plan.steps:
            step_result = await self.execute_recovery_step(step)

            if not step_result.success:
                await self.escalate_recovery_failure(step, step_result)
                break

            await self.update_recovery_status(incident.id, step.id, step_result)

        # Validate recovery
        validation_result = await self.validate_recovery(incident)

        return RecoveryResult(
            incident=incident,
            recovery_plan=recovery_plan,
            validation_result=validation_result,
            recovery_time=datetime.utcnow() - incident.started_at
        )

class BackupManager:
    """Manages all backup operations across the platform"""

    async def backup_agent_state(self, agent_id: str):
        """Create comprehensive backup of agent state and knowledge"""

        # Backup agent configuration
        agent_config = await self.agent_registry.get_agent_config(agent_id)

        # Backup agent memory and context
        agent_memory = await self.memory_manager.export_agent_memory(agent_id)

        # Backup learned patterns and optimizations
        agent_knowledge = await self.knowledge_manager.export_agent_knowledge(agent_

        # Create backup artifact
        backup_artifact = AgentBackup(
            agent_id=agent_id,
            timestamp=datetime.utcnow(),
            config=agent_config,
            memory=agent_memory,
            knowledge=agent_knowledge,
            version=await self.get_agent_version(agent_id)
        )
```

```
        # Store in multiple locations
        backup_locations = await self.store_backup_artifact(backup_artifact)

        return backup_locations

    async def restore_agent_from_backup(
        self,
        agent_id: str,
        backup_timestamp: datetime
    ):
        """Restore agent from backup to specific point in time"""

        # Find appropriate backup
        backup_artifact = await self.find_backup_artifact(
            agent_id, backup_timestamp
        )

        if not backup_artifact:
            raise BackupNotFoundError(
                f"No backup found for agent {agent_id} at {backup_timestamp}"
            )

        # Stop current agent
        await self.agent_manager.graceful_shutdown_agent(agent_id)

        # Restore from backup
        restored_agent = await self.restore_agent_from_artifact(backup_artifact)

        # Validate restoration
        validation_result = await self.validate_agent_restoration(restored_agent)

        if validation_result.is_valid:
            # Restart agent
            await self.agent_manager.start_agent(restored_agent)
            return restored_agent
        else:
            raise RestorationError(
                f"Agent restoration validation failed: {validation_result.errors}"
            )
```

2. High Availability Architecture

yaml

```yaml
# Kubernetes High Availability Configuration
apiVersion: v1
kind: ConfigMap
metadata:
  name: aamas-ha-config
data:
  ha-config.yaml: |
    high_availability:
      enabled: true
      replication_factor: 3
      auto_failover: true
      health_check_interval: 30s

    load_balancing:
      strategy: "round_robin"
      health_checks:
        - endpoint: "/health"
          interval: 10s
          timeout: 5s
          failure_threshold: 3

    data_replication:
      mode: "synchronous"
      regions: ["us-east-1", "us-west-2", "eu-west-1"]
      consistency_level: "strong"

    backup_schedule:
      frequency: "0 */4 * * *"  # Every 4 hours
      retention: "30d"
      verification: true

---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: aamas-database-cluster
spec:
  serviceName: aamas-db-service
  replicas: 3
  selector:
    matchLabels:
      app: aamas-database
  template:
```

```yaml
  metadata:
    labels:
      app: aamas-database
  spec:
    containers:
    - name: postgresql
      image: postgres:14-alpine
      env:
      - name: POSTGRES_DB
        value: "aamas"
      - name: POSTGRES_USER
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: username
      - name: POSTGRES_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: password
      - name: POSTGRES_REPLICATION_MODE
        value: "master"
      volumeMounts:
      - name: postgres-storage
        mountPath: /var/lib/postgresql/data
      ports:
      - containerPort: 5432
      livenessProbe:
        exec:
          command:
          - pg_isready
          - -U
          - $(POSTGRES_USER)
        initialDelaySeconds: 30
        periodSeconds: 10
      readinessProbe:
        exec:
          command:
          - pg_isready
          - -U
          - $(POSTGRES_USER)
        initialDelaySeconds: 5
        periodSeconds: 5
volumeClaimTemplates:
```

```
  - metadata:
      name: postgres-storage
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 100Gi
      storageClassName: "fast-ssd"
```

COST OPTIMIZATION

1. Intelligent Resource Management

```
python
```

```python
class CostOptimizationEngine:
    """AI-powered cost optimization for the platform"""

    def __init__(self):
        self.usage_analyzer = UsageAnalyzer()
        self.cost_predictor = CostPredictor()
        self.resource_optimizer = ResourceOptimizer()

    async def optimize_customer_costs(self, customer_id: str):
        """Analyze and optimize costs for a specific customer"""

        # Analyze current usage patterns
        usage_analysis = await self.usage_analyzer.analyze_customer_usage(
            customer_id, timeframe="30d"
        )

        # Predict future costs
        cost_prediction = await self.cost_predictor.predict_costs(
            customer_id, usage_analysis
        )

        # Generate optimization recommendations
        optimizations = await self.generate_cost_optimizations(
            usage_analysis, cost_prediction
        )

        # Auto-apply safe optimizations
        applied_optimizations = []
        for optimization in optimizations:
            if optimization.risk_level == "low" and optimization.auto_apply:
                result = await self.apply_optimization(customer_id, optimization)
                applied_optimizations.append(result)

        return CostOptimizationResult(
            current_costs=usage_analysis.total_cost,
            predicted_costs=cost_prediction.monthly_cost,
            potential_savings=sum(opt.estimated_savings for opt in optimizations),
            applied_optimizations=applied_optimizations,
            recommendations=optimizations
        )

    async def dynamic_resource_scaling(self):
        """Continuously optimize resource allocation across the platform"""
```

```
        while True:
            # Get current resource utilization
            utilization = await self.get_platform_utilization()

            # Identify optimization opportunities
            opportunities = await self.identify_scaling_opportunities(utilization)

            for opportunity in opportunities:
                if opportunity.action == "scale_down":
                    await self.scale_down_resources(opportunity)
                elif opportunity.action == "consolidate":
                    await self.consolidate_workloads(opportunity)
                elif opportunity.action == "spot_instance":
                    await self.migrate_to_spot_instances(opportunity)

            # Sleep for 5 minutes before next check
            await asyncio.sleep(300)

class ResourceOptimizer:
    """Optimizes compute resources for AI agents"""

    async def optimize_agent_resources(self, agent_id: str):
        """Optimize compute resources for a specific agent"""

        # Analyze agent performance history
        performance_history = await self.get_agent_performance_history(
            agent_id, timeframe="7d"
        )

        # Determine optimal resource allocation
        optimal_config = await self.calculate_optimal_resources(
            agent_id, performance_history
        )

        # Check if changes are beneficial
        cost_benefit = await self.analyze_cost_benefit(
            agent_id, optimal_config
        )

        if cost_benefit.net_benefit > 0:
            # Apply resource changes
            await self.apply_resource_changes(agent_id, optimal_config)
```

```
            return ResourceOptimizationResult(
                agent_id=agent_id,
                old_config=performance_history.current_config,
                new_config=optimal_config,
                estimated_savings=cost_benefit.monthly_savings,
                performance_impact=cost_benefit.performance_impact
            )

        return None  # No beneficial changes found
```

2. Cost Monitoring & Alerting

```
python
```

```python
class CostMonitoringSystem:
    """Real-time cost monitoring and alerting"""

    def __init__(self):
        self.cost_tracker = CostTracker()
        self.budget_manager = BudgetManager()
        self.alert_manager = AlertManager()

    async def setup_customer_cost_monitoring(self, customer_id: str):
        """Set up comprehensive cost monitoring for a customer"""

        # Set up cost tracking
        await self.cost_tracker.setup_customer_tracking(customer_id)

        # Configure budget alerts
        budget_config = await self.budget_manager.get_customer_budget(customer_id)

        alerts = [
            CostAlert(
                name=f"budget_50_percent_{customer_id}",
                condition=f"monthly_spend > {budget_config.monthly_limit * 0.5}",
                severity="info",
                message="You've reached 50% of your monthly budget"
            ),
            CostAlert(
                name=f"budget_80_percent_{customer_id}",
                condition=f"monthly_spend > {budget_config.monthly_limit * 0.8}",
                severity="warning",
                message="You've reached 80% of your monthly budget"
            ),
            CostAlert(
                name=f"budget_exceeded_{customer_id}",
                condition=f"monthly_spend > {budget_config.monthly_limit}",
                severity="critical",
                message="Monthly budget exceeded!"
            )
        ]

        for alert in alerts:
            await self.alert_manager.register_alert(alert)

    async def generate_cost_report(self, customer_id: str, period: str):
        """Generate detailed cost breakdown report"""
```

```
        cost_data = await self.cost_tracker.get_cost_breakdown(
            customer_id, period
        )

        # Analyze cost trends
        trends = await self.analyze_cost_trends(cost_data)

        # Generate optimization suggestions
        suggestions = await self.generate_cost_suggestions(cost_data, trends)

        return CostReport(
            customer_id=customer_id,
            period=period,
            total_cost=cost_data.total,
            breakdown=cost_data.breakdown,
            trends=trends,
            optimization_suggestions=suggestions,
            generated_at=datetime.utcnow()
        )
```

COMPLIANCE & GOVERNANCE

1. Data Governance Framework

```
python
```

```python
class DataGovernanceManager:
    """Manages data governance, privacy, and compliance"""

    def __init__(self):
        self.privacy_manager = PrivacyManager()
        self.compliance_checker = ComplianceChecker()
        self.audit_logger = AuditLogger()
        self.data_classifier = DataClassifier()

    async def setup_customer_governance(self, customer_id: str, requirements: Compli
        """Set up data governance for a customer based on their compliance needs"""

        # Classify customer data sensitivity
        data_classification = await self.data_classifier.classify_customer_data(
            customer_id
        )

        # Set up appropriate data handling policies
        policies = await self.generate_data_policies(
            data_classification, requirements
        )

        # Configure privacy controls
        await self.privacy_manager.setup_privacy_controls(
            customer_id, policies.privacy_policy
        )

        # Set up compliance monitoring
        await self.compliance_checker.setup_compliance_monitoring(
            customer_id, requirements, policies
        )

        # Configure audit logging
        await self.audit_logger.setup_customer_auditing(
            customer_id, policies.audit_policy
        )

        return GovernanceSetup(
            customer_id=customer_id,
            data_classification=data_classification,
            policies=policies,
            compliance_requirements=requirements
        )
```

```python
    async def handle_data_deletion_request(self, customer_id: str, deletion_request:
        """Handle GDPR/CCPA data deletion requests"""

        # Validate deletion request
        validation = await self.validate_deletion_request(deletion_request)
        if not validation.is_valid:
            raise InvalidDeletionRequest(validation.errors)

        # Find all customer data
        data_inventory = await self.create_data_inventory(
            customer_id, deletion_request.scope
        )

        # Execute deletion across all systems
        deletion_results = []
        for data_location in data_inventory.locations:
            result = await self.delete_data_from_location(
                data_location, deletion_request
            )
            deletion_results.append(result)

        # Verify deletion completeness
        verification = await self.verify_data_deletion(
            customer_id, deletion_request, deletion_results
        )

        # Log deletion for compliance
        await self.audit_logger.log_data_deletion(
            customer_id=customer_id,
            deletion_request=deletion_request,
            results=deletion_results,
            verification=verification
        )

        return DataDeletionResult(
            request=deletion_request,
            results=deletion_results,
            verification=verification,
            completed_at=datetime.utcnow()
        )

class ComplianceFramework:
    """Comprehensive compliance framework supporting multiple standards"""
```

```
supported_standards = [
    "SOC2_TYPE2",
    "ISO27001",
    "GDPR",
    "CCPA",
    "HIPAA",
    "PCI_DSS"
]

async def assess_compliance_posture(self, customer_id: str):
    """Assess current compliance posture across all applicable standards"""

    customer_profile = await self.get_customer_compliance_profile(customer_id)

    assessments = {}
    for standard in customer_profile.applicable_standards:
        assessment = await self.assess_standard_compliance(
            customer_id, standard
        )
        assessments[standard] = assessment

    # Generate overall compliance score
    overall_score = self.calculate_overall_compliance_score(assessments)

    # Identify critical gaps
    critical_gaps = self.identify_critical_gaps(assessments)

    # Generate remediation plan
    remediation_plan = await self.generate_remediation_plan(critical_gaps)

    return ComplianceAssessment(
        customer_id=customer_id,
        overall_score=overall_score,
        standard_assessments=assessments,
        critical_gaps=critical_gaps,
        remediation_plan=remediation_plan,
        next_assessment_date=datetime.utcnow() + timedelta(days=90)
    )
```

CONCLUSION & NEXT STEPS

This comprehensive system architecture provides AAMAS with:

✅ Technical Foundation

- **Scalable microservices architecture** supporting multi-tenant SaaS operations
- **Intelligent agent orchestration** with sophisticated communication protocols
- **Enterprise-grade security** with zero-trust principles and comprehensive auditing
- **High availability & disaster recovery** with multi-region redundancy

✅ Operational Excellence

- **Real-time monitoring & observability** with predictive alerting
- **Cost optimization** through AI-powered resource management
- **Compliance framework** supporting major industry standards
- **Developer-friendly APIs** with both REST and GraphQL interfaces

✅ Business Enablement

- **Multi-tier pricing** support with granular resource allocation
- **Customer analytics** with actionable insights and ROI tracking
- **Integration ecosystem** supporting 50+ development tools
- **White-label capabilities** for enterprise customization

🚀 Implementation Roadmap

Phase 1 (Months 1-3): Core Infrastructure

- Deploy basic microservices architecture
- Implement agent communication framework
- Set up development/staging environments
- Basic security and monitoring

Phase 2 (Months 4-6): Agent Intelligence

- Deploy specialized agent implementations
- Integrate AI models and knowledge systems
- Implement predictive analytics
- Customer onboarding system

Phase 3 (Months 7-9): Enterprise Features

- Advanced security and compliance
- Multi-region deployment
- Cost optimization engine
- Advanced analytics and reporting

Phase 4 (Months 10-12): Scale & Optimize

- Performance optimization
- Advanced integrations
- Global expansion features
- AI model fine-tuning

This architecture positions AAMAS as a truly enterprise-ready platform capable of transforming how software development teams operate, with the intelligence and automation needed to achieve the ambitious business goals outlined in your proposal.