YTEMPIRE Capacity Planning Documents

Version 1.0 - Resource Projections & Performance Models

Table of Contents

Executive Summary

This Capacity Planning document provides comprehensive resource projections and performance models for YTEMPIRE's growth from initial deployment to enterprise scale. The planning methodology ensures optimal resource utilization while maintaining performance targets and controlling costs through intelligent scaling strategies.

Key Planning Insights:

- **Resource Efficiency**: Achieve 85% resource utilization through intelligent scheduling and auto-scaling
- **Cost Optimization**: Reduce infrastructure costs by 65% compared to traditional scaling approaches
- **Performance Guarantee**: Maintain sub-second response times even at 100x current scale
- **Storage Efficiency**: Implement tiered storage reducing costs by 70% while maintaining accessibility

- **Predictive Scaling**: AI-driven capacity predictions with 95% accuracy for proactive resource allocation

Projected Growth Trajectory:

- Month 1-3: 2 channels, 6 videos/day, 10GB/day data growth
- Month 4-6: 10 channels, 30 videos/day, 100GB/day data growth
- Month 7-12: 50 channels, 150 videos/day, 500GB/day data growth
- Year 2+: 100+ channels, 300+ videos/day, 1TB+/day data growth

Cost Projections:

- Initial Phase (Local): $500/month (hardware amortization + utilities)
- Growth Phase (Hybrid): $2,500/month (cloud services + local resources)
- Scale Phase (Cloud): $8,000/month (optimized cloud infrastructure)
- Enterprise Phase: $15,000/month (global multi-region deployment)

Resource Projections

Compute Resource Requirements

Current State Analysis (Local Deployment)

```python
class ComputeResourceCalculator:
    """Calculate compute resource requirements based on workload"""

    def __init__(self):
        self.baseline_requirements = {
            'cpu_per_video': 2.5,  # CPU cores per video processing
            'gpu_hours_per_video': 0.25,  # GPU hours per video
            'memory_per_video': 4,  # GB RAM per video processing
            'concurrent_processing': 3,  # Max concurrent video processing

            'service_overhead': {
                'orchestrator': {'cpu': 1, 'memory': 2},
                'trend_analyzer': {'cpu': 2, 'memory': 4},
                'content_generator': {'cpu': 2, 'memory': 8},
                'media_processor': {'cpu': 4, 'memory': 16},
                'publisher': {'cpu': 1, 'memory': 2},
                'analytics': {'cpu': 2, 'memory': 4}
            }
```

Container Resource Optimization

yaml

```yaml
container_optimization:
  cpu_optimization:
    technique: "CPU pinning and NUMA awareness"
    implementation:
      node_affinity:
        - key: workload-type
          values: [cpu-intensive]
      cpu_manager_policy: static
      topology_manager_policy: best-effort

    resource_limits:
      guaranteed:
        cpu: "2"
        memory: "4Gi"
      burstable:
        cpu: "4"
        memory: "8Gi"

  memory_optimization:
    techniques:
      - huge_pages: enabled
      - memory_compression: zram
      - swap_accounting: disabled

    jvm_tuning:  # For Java services
      heap_size: "75% of container memory"
      gc_algorithm: "G1GC"
      gc_flags: "-XX:+UseG1GC -XX:MaxGCPauseMillis=200"

  gpu_optimization:
    sharing_strategy: "MPS"  # Multi-Process Service
    memory_allocation: "dynamic"
    scheduling:
      - exclusive_processes: [training]
      - shared_processes: [inference]
    utilization_target: 85%

  network_optimization:
    techniques:
      - sr_iov: enabled
      - cpu_affinity: "network interrupts"
      - buffer_tuning:
          rmem_max: 134217728
          wmem_max: 134217728
```

Spot Instance Strategy

python

```python
class SpotInstanceOptimizer:
    """Optimize usage of spot instances for cost savings"""

    def __init__(self):
        self.spot_strategies = {
            'diversified': {
                'instance_types': [
                    'c6i.large', 'c6i.xlarge', 'c6i.2xlarge',
                    'c6a.large', 'c6a.xlarge', 'c6a.2xlarge',
                    'c5.large', 'c5.xlarge', 'c5.2xlarge'
                ],
                'availability_zones': ['us-east-1a', 'us-east-1b', 'us-east-1c'],
                'allocation_strategy': 'capacity-optimized'
            },
            'workload_mapping': {
                'stateless': {'spot_percentage': 80},
                'batch_processing': {'spot_percentage': 90},
                'critical_services': {'spot_percentage': 0},
                'gpu_workloads': {'spot_percentage': 60}
            }
        }

        self.interruption_handling = {
            'grace_period': 120,  # seconds
            'checkpointing': True,
            'state_preservation': 's3',
            'relaunch_strategy': 'immediate'
        }

    def calculate_spot_savings(self, workload: dict) -> dict:
        """Calculate potential savings from spot instances"""

        on_demand_cost = 0
        spot_cost = 0

        for instance_type, hours in workload['instance_hours'].items():
            on_demand_price = self.get_on_demand_price(instance_type)
            spot_price = self.get_spot_price(instance_type)

            on_demand_cost += on_demand_price * hours

            # Apply spot percentage based on workload type
            spot_percentage = self.spot_strategies['workload_mapping'].get(
                workload['type'], {}
            ).get('spot_percentage', 0) / 100
```

```
        spot_hours = hours * spot_percentage
        on_demand_hours = hours * (1 - spot_percentage)

        spot_cost += (spot_price * spot_hours + on_demand_price * on_demand_hours)

    savings = on_demand_cost - spot_cost
    savings_percentage = (savings / on_demand_cost) * 100 if on_demand_cost > 0 else 0

    return {
        'on_demand_cost': on_demand_cost,
        'optimized_cost': spot_cost,
        'savings': savings,
        'savings_percentage': savings_percentage,
        'risk_score': self.calculate_interruption_risk(workload)
    }
```

Cost Optimization Recommendations

Comprehensive Cost Optimization Framework

python

```python
class CostOptimizationFramework:
    """Framework for identifying and implementing cost optimizations"""

    def __init__(self):
        self.optimization_categories = {
            'compute': ComputeCostOptimizer(),
            'storage': StorageCostOptimizer(),
            'network': NetworkCostOptimizer(),
            'services': ServiceCostOptimizer()
        }

        self.optimization_strategies = {
            'immediate': {  # Can be implemented immediately
                'reserved_instances': {
                    'potential_savings': '35-72%',
                    'commitment': '1-3 years',
                    'risk': 'low'
                },
                'spot_instances': {
                    'potential_savings': '50-90%',
                    'commitment': 'none',
                    'risk': 'medium'
                },
                'right_sizing': {
                    'potential_savings': '15-40%',
                    'commitment': 'none',
                    'risk': 'low'
                }
            },
            'short_term': {  # 1-3 months
                'auto_scaling': {
                    'potential_savings': '20-50%',
                    'implementation_time': '2 weeks',
                    'complexity': 'medium'
                },
                'storage_tiering': {
                    'potential_savings': '40-70%',
                    'implementation_time': '1 month',
                    'complexity': 'medium'
                },
                'cdn_optimization': {
                    'potential_savings': '30-60%',
                    'implementation_time': '2 weeks',
                    'complexity': 'low'
                }
            },
```

```
            'long_term': {  # 3-6 months
                'architecture_optimization': {
                    'potential_savings': '40-60%',
                    'implementation_time': '3-6 months',
                    'complexity': 'high'
                },
                'serverless_migration': {
                    'potential_savings': '50-80%',
                    'implementation_time': '4-6 months',
                    'complexity': 'high'
                }
            }
        }

    async def generate_optimization_plan(self, current_costs: dict) -> dict:
        """Generate comprehensive cost optimization plan"""

        # Analyze current spending
        cost_analysis = await self._analyze_current_costs(current_costs)

        # Identify optimization opportunities
        opportunities = []

        for category, optimizer in self.optimization_categories.items():
            category_opportunities = await optimizer.identify_opportunities(
                cost_analysis[category]
            )
            opportunities.extend(category_opportunities)

        # Prioritize opportunities
        prioritized_opportunities = self._prioritize_opportunities(
            opportunities,
            factors={
                'savings_potential': 0.4,
                'implementation_ease': 0.3,
                'risk_level': 0.2,
                'time_to_value': 0.1
            }
        )

        # Create implementation roadmap
        roadmap = self._create_implementation_roadmap(prioritized_opportunities)

        # Calculate total potential savings
        total_savings = sum(opp['estimated_savings'] for opp in prioritized_opportunities)
        roi_timeline = self._calculate_roi_timeline(roadmap, current_costs)
```

```
    return {
        'current_monthly_cost': sum(current_costs.values()),
        'optimization_opportunities': prioritized_opportunities,
        'implementation_roadmap': roadmap,
        'total_potential_savings': total_savings,
        'savings_percentage': (total_savings / sum(current_costs.values())) * 100,
        'roi_timeline': roi_timeline,
        'quick_wins': [opp for opp in prioritized_opportunities
                       if opp['implementation_time'] < 7],
        'risk_assessment': self._assess_optimization_risks(prioritized_opportunities)
    }
```

Specific Cost Optimization Tactics

yaml

```yaml
cost_optimization_tactics:
  compute_optimization:
    reserved_instances:
      recommendation: "70% baseline as RIs"
      types:
        - standard_ri: "For predictable workloads"
        - convertible_ri: "For flexibility"
      savings: "35-72% vs on-demand"

    savings_plans:
      compute_savings_plan: "1-year, 50% commitment"
      ec2_instance_savings_plan: "3-year for stable workloads"
      coverage_target: 60%

    spot_fleet:
      configuration:
        - allocation_strategy: "capacity-optimized"
        - instance_pools: 10
        - target_capacity: "40% of total"
      use_cases:
        - batch_processing
        - stateless_services
        - development_environments

  storage_optimization:
    s3_lifecycle_policies:
      - name: "video-archives"
        transitions:
          - days: 30
            storage_class: "STANDARD_IA"
          - days: 90
            storage_class: "GLACIER"
          - days: 365
            storage_class: "DEEP_ARCHIVE"

    intelligent_tiering:
      enabled_for:
        - unknown_access_patterns
        - large_datasets
      monitoring_charges: "$0.0025 per 1,000 objects"

    ebs_optimization:
      - gp3_migration: "Save 20% vs gp2"
      - snapshot_lifecycle: "Delete after 30 days"
      - unused_volume_cleanup: "Weekly automation"
```

```yaml
network_optimization:
  cloudfront_caching:
    cache_behaviors:
      - path_pattern: "*.mp4"
        ttl: 86400
      - path_pattern: "*.jpg"
        ttl: 604800
    origin_shield: enabled
    compression: enabled

  vpc_endpoints:
    services:
      - s3: "Save data transfer costs"
      - dynamodb: "Reduce NAT gateway usage"
    estimated_savings: "$500-2000/month"

  nat_instance_vs_gateway:
    recommendation: "NAT instance for dev/test"
    savings: "90% for low-traffic environments"

service_optimization:
  api_caching:
    strategy: "Cache all read operations"
    ttl_configuration:
      - youtube_data: 300  # 5 minutes
      - analytics: 3600    # 1 hour
      - static_data: 86400 # 24 hours
    estimated_reduction: "70% API calls"

  batch_processing:
    recommendation: "Batch API calls"
    batch_sizes:
      - youtube_api: 50
      - analytics_api: 100
    frequency: "Every 5 minutes"

  model_optimization:
    quantization: "Reduce model size by 75%"
    pruning: "Remove 50% of parameters"
    distillation: "Use smaller models for inference"
```

Cost Monitoring and Alerting

python

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

    def __init__(self):
        self.cost_thresholds = {
            'daily': {
                'warning': 1.2,  # 20% over average
                'critical': 1.5  # 50% over average
            },
            'service': {
                'compute': 5000,  # $/month
                'storage': 2000,
                'network': 1000,
                'apis': 1500
            }
        }

        self.anomaly_detector = CostAnomalyDetector()
        self.alert_manager = CostAlertManager()

    async def monitor_costs(self) -> dict:
        """Monitor costs in real-time"""

        # Get current costs
        current_costs = await self._fetch_current_costs()

        # Detect anomalies
        anomalies = await self.anomaly_detector.detect(current_costs)

        # Check thresholds
        threshold_violations = self._check_thresholds(current_costs)

        # Generate alerts
        if anomalies or threshold_violations:
            alerts = await self.alert_manager.create_alerts(
                anomalies + threshold_violations
            )

            # Take automated actions
            for alert in alerts:
                if alert['severity'] == 'critical':
                    await self._take_automated_action(alert)

        # Generate cost report
        report = {
            'current_daily_cost': current_costs['daily_total'],
```

```
            'projected_monthly_cost': current_costs['daily_total'] * 30,
            'cost_by_service': current_costs['by_service'],
            'anomalies_detected': anomalies,
            'threshold_violations': threshold_violations,
            'automated_actions': self._get_automated_actions(),
            'savings_opportunities': await self._identify_immediate_savings()
        }

        return report

    async def _take_automated_action(self, alert: dict):
        """Take automated action for critical cost alerts"""

        if alert['type'] == 'unexpected_spike':
            # Scale down non-critical services
            await self._scale_down_non_critical()

        elif alert['type'] == 'quota_approaching':
            # Implement rate limiting
            await self._implement_emergency_rate_limiting()

        elif alert['type'] == 'budget_exceeded':
            # Switch to cost-saving mode
            await self._enable_cost_saving_mode()
```

Implementation Guidelines

Capacity Planning Process

yaml

```yaml
capacity_planning_process:
  phase_1_assessment:
    duration: "1 week"
    activities:
      - current_state_analysis
      - workload_characterization
      - baseline_establishment
      - bottleneck_identification
    deliverables:
      - capacity_assessment_report
      - performance_baseline_document
      - bottleneck_analysis

  phase_2_modeling:
    duration: "2 weeks"
    activities:
      - workload_modeling
      - growth_projection
      - scenario_planning
      - cost_modeling
    deliverables:
      - capacity_model
      - growth_scenarios
      - cost_projections

  phase_3_optimization:
    duration: "2 weeks"
    activities:
      - optimization_analysis
      - architecture_review
      - tool_evaluation
      - poc_testing
    deliverables:
      - optimization_recommendations
      - architecture_improvements
      - tool_selection

  phase_4_implementation:
    duration: "4 weeks"
    activities:
      - infrastructure_updates
      - monitoring_setup
      - automation_implementation
      - testing_validation
    deliverables:
      - updated_infrastructure
```

- monitoring_dashboards
- automation_scripts
- test_results

Continuous Capacity Management

python

```python
class ContinuousCapacityManagement:
    """Implement continuous capacity management practices"""

    def __init__(self):
        self.review_cycle = {
            'daily': ['utilization_check', 'anomaly_detection'],
            'weekly': ['trend_analysis', 'forecast_update'],
            'monthly': ['capacity_review', 'optimization_assessment'],
            'quarterly': ['architecture_review', 'strategic_planning']
        }

        self.automation_rules = {
            'auto_scaling': {
                'enabled': True,
                'min_capacity': 0.3,
                'max_capacity': 0.85,
                'scale_up_threshold': 0.7,
                'scale_down_threshold': 0.3
            },
            'predictive_scaling': {
                'enabled': True,
                'prediction_window': 3600,  # 1 hour
                'ml_model': 'time_series_forecast'
            }
        }

    async def execute_daily_review(self) -> dict:
        """Execute daily capacity review"""

        review_results = {
            'date': datetime.utcnow().date(),
            'metrics': {},
            'issues': [],
            'actions': []
        }

        # Check utilization
        utilization = await self._check_utilization()
        review_results['metrics']['utilization'] = utilization

        # Detect anomalies
        anomalies = await self._detect_anomalies()
        if anomalies:
            review_results['issues'].extend(anomalies)

        # Check scaling triggers
```

```
        scaling_needed = await self._evaluate_scaling_needs(utilization)
        if scaling_needed:
            action = await self._execute_scaling(scaling_needed)
            review_results['actions'].append(action)

        # Update forecasts
        await self._update_short_term_forecast()

        return review_results
```

Capacity Planning Tools

yaml

```yaml
recommended_tools:
  monitoring:
    - name: "Prometheus + Grafana"
      purpose: "Metrics collection and visualization"
      features:
        - real_time_monitoring
        - custom_dashboards
        - alerting

    - name: "Datadog"
      purpose: "Unified monitoring platform"
      features:
        - apm_integration
        - log_analytics
        - ml_powered_insights

  load_testing:
    - name: "K6"
      purpose: "Performance testing"
      features:
        - scriptable_tests
        - cloud_execution
        - ci_integration

    - name: "Gatling"
      purpose: "High-performance load testing"
      features:
        - scala_dsl
        - detailed_reports
        - distributed_testing

  capacity_planning:
    - name: "Kubernetes VPA/HPA"
      purpose: "Automatic scaling"
      features:
        - vertical_scaling
        - horizontal_scaling
        - custom_metrics

    - name: "AWS Compute Optimizer"
      purpose: "Right-sizing recommendations"
      features:
        - ml_recommendations
        - cost_optimization
        - performance_analysis
```

Monitoring and Adjustment Framework

Capacity Metrics Dashboard

yaml

```yaml
capacity_dashboard:
  real_time_metrics:
    - metric: cpu_utilization
      threshold_warning: 70%
      threshold_critical: 85%

    - metric: memory_utilization
      threshold_warning: 75%
      threshold_critical: 90%

    - metric: gpu_utilization
      threshold_warning: 80%
      threshold_critical: 95%

    - metric: storage_usage
      threshold_warning: 70%
      threshold_critical: 85%

    - metric: network_throughput
      threshold_warning: 80%
      threshold_critical: 95%

  trend_analysis:
    - metric: daily_growth_rate
      calculation: "7-day moving average"
      alert_threshold: "> 10%"

    - metric: peak_utilization_trend
      calculation: "30-day trend"
      alert_threshold: "increasing > 5% week-over-week"

  predictive_indicators:
    - metric: days_until_capacity
      calculation: "current_growth_rate"
      alert_threshold: "< 30 days"

    - metric: cost_efficiency
      calculation: "cost_per_video"
      alert_threshold: "> $3"
```

Automated Capacity Adjustments

python

```python
class AutomatedCapacityAdjustment:
    """Automated capacity adjustment based on real-time metrics"""

    def __init__(self):
        self.adjustment_policies = {
            'conservative': {
                'scale_up_buffer': 0.3,
                'scale_down_delay': 1800,
                'min_instances': 3
            },
            'aggressive': {
                'scale_up_buffer': 0.1,
                'scale_down_delay': 300,
                'min_instances': 1
            },
            'cost_optimized': {
                'scale_up_buffer': 0.2,
                'scale_down_delay': 900,
                'prefer_spot': True
            }
        }

        self.ml_predictor = CapacityPredictor()
        self.cost_analyzer = CostAnalyzer()

    async def adjust_capacity(self, current_metrics: dict) -> dict:
        """Make capacity adjustments based on current state"""

        # Predict future demand
        demand_forecast = await self.ml_predictor.predict(
            current_metrics,
            horizon_minutes=60
        )

        # Determine required capacity
        required_capacity = self._calculate_required_capacity(
            demand_forecast,
            self.adjustment_policies['cost_optimized']
        )

        # Compare with current capacity
        current_capacity = current_metrics['total_capacity']
        adjustment_needed = required_capacity - current_capacity

        # Make adjustment decision
        if abs(adjustment_needed) > current_capacity * 0.1:  # 10% threshold
```

```
            adjustment_plan = await self._create_adjustment_plan(
                adjustment_needed,
                current_metrics
            )

            # Execute adjustment
            result = await self._execute_adjustment(adjustment_plan)

            return {
                'adjustment_made': True,
                'adjustment_type': 'scale_up' if adjustment_needed > 0 else 'scale_down',
                'adjustment_amount': adjustment_needed,
                'new_capacity': required_capacity,
                'execution_result': result,
                'estimated_cost_impact': await self.cost_analyzer.estimate_impact(
                    adjustment_plan
                )
            }

        return {
            'adjustment_made': False,
            'reason': 'Within acceptable threshold',
            'current_capacity': current_capacity,
            'required_capacity': required_capacity
        }
```

Conclusion

This Capacity Planning document provides a comprehensive framework for managing YTEMPIRE's infrastructure growth from initial deployment to global scale. The key achievements include:

1. **Resource Efficiency**: 85% utilization through intelligent scheduling and auto-scaling
2. **Cost Optimization**: 65% reduction compared to traditional approaches
3. **Performance Guarantee**: Sub-second response times maintained at 100x scale
4. **Predictive Scaling**: 95% accuracy in capacity predictions
5. **Automated Management**: Self-adjusting infrastructure based on real-time metrics

The planning models and optimization strategies ensure YTEMPIRE can scale efficiently while maintaining performance targets and controlling costs. The implementation of these capacity planning practices will enable seamless growth from 2 to 100+ channels without architectural limitations.

Document Version: 1.0
Last Updated: [Current Date]

Author: Saad T. - Solution Architect }

```
def calculate_requirements(self, channels: int, videos_per_day: int) -> dict:
    """Calculate total compute requirements"""

    # Video processing requirements
    daily_cpu_hours = videos_per_day * self.baseline_requirements['cpu_per_video']
    daily_gpu_hours = videos_per_day * self.baseline_requirements['gpu_hours_per_video']
    peak_memory = self.baseline_requirements['concurrent_processing'] * \
                  self.baseline_requirements['memory_per_video']

    # Service overhead
    service_cpu = sum(s['cpu'] for s in self.baseline_requirements['service_overhead'].values())
    service_memory = sum(s['memory'] for s in self.baseline_requirements['service_overhead'].values())

    # Total requirements with 20% buffer
    total_requirements = {
        'cpu_cores': math.ceil((daily_cpu_hours / 24 + service_cpu) * 1.2),
        'gpu_hours_daily': daily_gpu_hours,
        'memory_gb': math.ceil((peak_memory + service_memory) * 1.2),
        'gpu_memory_gb': math.ceil(videos_per_day * 0.5),  # 0.5GB GPU memory per video
    }

    return total_requirements
```

Compute Scaling Projections

```yaml
compute_scaling_timeline:
  phase_1_local:  # Months 1-3
    infrastructure: "Single Workstation"
    specifications:
      cpu: "AMD Ryzen 9 7950X3D (16 cores)"
      gpu: "NVIDIA RTX 5090 (32GB)"
      memory: "128GB DDR5"
    capacity:
      channels: 2
      videos_per_day: 6
      concurrent_workflows: 3
    utilization:
      cpu_average: 45%
      gpu_average: 30%
      memory_peak: 40%

  phase_2_hybrid:  # Months 4-6
    infrastructure: "Local + Cloud Burst"
    local_resources:
      cpu: "AMD Ryzen 9 7950X3D (16 cores)"
      gpu: "2x NVIDIA RTX 5090 (64GB total)"
      memory: "256GB DDR5"
    cloud_resources:
      instances:
        - type: "c6i.4xlarge"
          count: 2
          purpose: "CPU intensive tasks"
        - type: "g5.2xlarge"
          count: 1
          purpose: "GPU overflow"
    capacity:
      channels: 10
      videos_per_day: 30
      concurrent_workflows: 8

  phase_3_cloud_primary:  # Months 7-12
    infrastructure: "Cloud-First with Local Cache"
    aws_resources:
      compute:
        - type: "c6i.8xlarge"
          count: 3
          autoscaling: { min: 2, max: 5 }
```

```
      - type: "g5.4xlarge"
        count: 2
        autoscaling: { min: 1, max: 4 }
    kubernetes:
      cluster: "eks"
      nodes: 5-15
      gpu_nodes: 2-4
  capacity:
    channels: 50
    videos_per_day: 150
    concurrent_workflows: 30

phase_4_multi_region:  # Year 2+
  infrastructure: "Global Multi-Region"
  regions:
    primary: "us-east-1"
    secondary: ["eu-west-1", "ap-southeast-1"]
  resources_per_region:
    compute_nodes: 10-50
    gpu_nodes: 5-20
    edge_locations: 25
  capacity:
    channels: 100+
    videos_per_day: 300+
    concurrent_workflows: 100+
```

Service-Specific Compute Requirements

python

```python
class ServiceComputeModel:
    """Model compute requirements for each service"""

    def __init__(self):
        self.service_models = {
            'orchestrator': {
                'cpu_per_request': 0.01,  # CPU seconds per request
                'memory_per_connection': 10,  # MB per connection
                'baseline_cpu': 0.5,
                'baseline_memory': 1024,  # MB
                'scaling_factor': 1.5
            },
            'trend_analyzer': {
                'cpu_per_analysis': 2.0,  # CPU seconds per trend analysis
                'memory_per_dataset': 500,  # MB per dataset
                'ml_model_memory': 2048,  # MB for ML models
                'gpu_utilization': 0.2,  # 20% GPU usage
                'scaling_factor': 2.0
            },
            'content_generator': {
                'cpu_per_script': 5.0,  # CPU seconds per script
                'memory_per_generation': 1024,  # MB per generation
                'llm_memory': 4096,  # MB for LLM models
                'gpu_utilization': 0.4,  # 40% GPU usage
                'scaling_factor': 2.5
            },
            'media_processor': {
                'cpu_per_minute_video': 60,  # CPU seconds per minute of video
                'memory_per_video': 4096,  # MB per video processing
                'gpu_per_minute_video': 30,  # GPU seconds per minute
                'gpu_memory_per_video': 8192,  # MB GPU memory
                'scaling_factor': 3.0
            }
        }

    def project_service_requirements(self, service: str, load: dict) -> dict:
        """Project compute requirements for a service under given load"""

        model = self.service_models[service]

        if service == 'orchestrator':
            cpu_required = (load['requests_per_second'] * model['cpu_per_request'] +
                            model['baseline_cpu']) * model['scaling_factor']
            memory_required = (load['concurrent_connections'] * model['memory_per_connection'] +
                               model['baseline_memory']) * model['scaling_factor']
```

```
    elif service == 'trend_analyzer':
        cpu_required = (load['analyses_per_hour'] / 3600 * model['cpu_per_analysis'] +
                        model['baseline_cpu']) * model['scaling_factor']
        memory_required = (load['active_datasets'] * model['memory_per_dataset'] +
                           model['ml_model_memory']) * model['scaling_factor']

    # ... similar calculations for other services

    return {
        'cpu_cores': math.ceil(cpu_required),
        'memory_mb': math.ceil(memory_required),
        'gpu_utilization': model.get('gpu_utilization', 0),
        'instances_required': math.ceil(cpu_required / 4)  # Assuming 4 cores per instance
    }
```

Storage Growth Projections

Storage Requirements Model

python

```python
class StorageGrowthModel:
    """Model storage growth across all components"""

    def __init__(self):
        self.storage_factors = {
            'video_content': {
                'raw_video_per_minute': 500,  # MB per minute
                'processed_video_per_minute': 50,  # MB per minute (compressed)
                'thumbnail_size': 2,  # MB per thumbnail
                'thumbnails_per_video': 10,  # A/B testing variants
                'retention_days': {
                    'raw': 7,
                    'processed': 365,
                    'thumbnails': 365
                }
            },
            'ml_models': {
                'llm_models': 50000,  # MB total
                'vision_models': 20000,  # MB total
                'custom_models': 10000,  # MB total
                'model_versions': 5  # Keep 5 versions
            },
            'analytics_data': {
                'metrics_per_video': 10,  # MB of analytics data
                'aggregated_daily': 100,  # MB per day
                'retention_days': 730  # 2 years
            },
            'database': {
                'growth_per_video': 5,  # MB per video metadata
                'indexes_overhead': 1.3,  # 30% overhead for indexes
                'backup_copies': 3
            }
        }

    def project_storage_growth(self, timeline_months: int) -> dict:
        """Project storage growth over timeline"""

        projections = []

        for month in range(1, timeline_months + 1):
            # Calculate videos produced
            if month <= 3:
                daily_videos = 6
                channels = 2
            elif month <= 6:
                daily_videos = 30
```

```
            channels = 10
        elif month <= 12:
            daily_videos = 150
            channels = 50
        else:
            daily_videos = 300
            channels = 100

        monthly_videos = daily_videos * 30
        total_videos_to_date = sum([
            6 * 30 * min(month, 3),  # First 3 months
            30 * 30 * max(0, min(month - 3, 3)),  # Months 4-6
            150 * 30 * max(0, min(month - 6, 6)),  # Months 7-12
            300 * 30 * max(0, month - 12)  # After 12 months
        ])

        # Calculate storage requirements
        video_storage = self._calculate_video_storage(monthly_videos, total_videos_to_date)
        ml_storage = self.storage_factors['ml_models']['llm_models'] + \
                     self.storage_factors['ml_models']['vision_models'] + \
                     self.storage_factors['ml_models']['custom_models']
        analytics_storage = total_videos_to_date * self.storage_factors['analytics_data']['metrics_per_video']
        database_storage = total_videos_to_date * self.storage_factors['database']['growth_per_video'] * \
                           self.storage_factors['database']['indexes_overhead']

        total_storage = video_storage + ml_storage + analytics_storage + database_storage

        projections.append({
            'month': month,
            'channels': channels,
            'daily_videos': daily_videos,
            'total_videos': total_videos_to_date,
            'storage_breakdown': {
                'video_content': video_storage,
                'ml_models': ml_storage,
                'analytics': analytics_storage,
                'database': database_storage,
                'total_gb': total_storage / 1024
            }
        })

    return projections
```

Storage Tier Strategy

yaml

```yaml
storage_tier_strategy:
  hot_tier:
    description: "Active content and real-time data"
    storage_type: "NVMe SSD"
    retention: "0-30 days"
    access_pattern: "Frequent random access"
    components:
      - active_video_projects
      - recent_thumbnails
      - live_analytics
      - cache_data
    local_capacity: "4TB"
    cloud_service: "EBS gp3"
    cost_per_gb_month: "$0.08"

  warm_tier:
    description: "Recent content and analytics"
    storage_type: "SSD"
    retention: "31-90 days"
    access_pattern: "Occasional access"
    components:
      - processed_videos
      - historical_analytics
      - ml_training_data
    local_capacity: "16TB"
    cloud_service: "EBS gp2"
    cost_per_gb_month: "$0.05"

  cold_tier:
    description: "Archive and compliance"
    storage_type: "HDD"
    retention: "91-365 days"
    access_pattern: "Rare access"
    components:
      - archived_videos
      - compliance_data
      - old_analytics
    local_capacity: "48TB"
    cloud_service: "S3 Standard-IA"
    cost_per_gb_month: "$0.0125"

  archive_tier:
    description: "Long-term retention"
    storage_type: "Tape/Glacier"
    retention: "365+ days"
    access_pattern: "Very rare access"
```

```
  components:
    - compliance_archives
    - historical_backups
  cloud_service: "S3 Glacier Deep Archive"
  cost_per_gb_month: "$0.00099"
```

Storage Growth Visualization

```python
def generate_storage_projection_chart():
    """Generate storage growth projection chart"""

    months = range(1, 25)
    storage_types = {
        'video_content': [],
        'ml_models': [],
        'analytics': [],
        'database': [],
        'total': []
    }

    for month in months:
        # Simplified calculation for visualization
        video_growth = 500 * (1.5 ** (month / 6))  # Exponential growth
        ml_constant = 100  # Relatively constant
        analytics_growth = 50 * month  # Linear growth
        database_growth = 20 * month * 1.2  # Linear with overhead

        storage_types['video_content'].append(video_growth)
        storage_types['ml_models'].append(ml_constant)
        storage_types['analytics'].append(analytics_growth)
        storage_types['database'].append(database_growth)
        storage_types['total'].append(video_growth + ml_constant + analytics_growth + database_growth)

    return {
        'months': list(months),
        'projections': storage_types,
        'total_year_1': storage_types['total'][11],
        'total_year_2': storage_types['total'][23]
    }
```

Network Bandwidth Analysis

Bandwidth Requirements Model

python

```python
class NetworkBandwidthModel:
    """Model network bandwidth requirements"""

    def __init__(self):
        self.bandwidth_factors = {
            'video_upload': {
                'average_size_mb': 500,
                'upload_duration_seconds': 300,
                'concurrent_uploads': 3
            },
            'api_traffic': {
                'youtube_api': {
                    'requests_per_video': 10,
                    'avg_request_size_kb': 5,
                    'avg_response_size_kb': 20
                },
                'ai_services': {
                    'requests_per_video': 20,
                    'avg_request_size_kb': 50,
                    'avg_response_size_kb': 100
                }
            },
            'internal_traffic': {
                'service_mesh': {
                    'requests_per_second': 100,
                    'avg_payload_kb': 10
                },
                'database_replication': {
                    'bandwidth_mbps': 10
                },
                'monitoring': {
                    'metrics_bandwidth_mbps': 5,
                    'logs_bandwidth_mbps': 10
                }
            },
            'cdn_traffic': {
                'thumbnail_serving': {
                    'requests_per_minute': 1000,
                    'avg_size_kb': 200
                },
                'analytics_dashboard': {
                    'concurrent_users': 10,
                    'bandwidth_per_user_mbps': 2
                }
            }
        }
```

```python
def calculate_bandwidth_requirements(self, scale_factor: dict) -> dict:
    """Calculate total bandwidth requirements"""

    # Upload bandwidth
    upload_bandwidth = (
        self.bandwidth_factors['video_upload']['average_size_mb'] * 8 *
        scale_factor['concurrent_uploads'] /
        self.bandwidth_factors['video_upload']['upload_duration_seconds']
    )

    # API bandwidth
    api_bandwidth = 0
    for service, params in self.bandwidth_factors['api_traffic'].items():
        requests_per_second = (params['requests_per_video'] *
                               scale_factor['videos_per_hour'] / 3600)
        bandwidth_mbps = (requests_per_second *
                          (params['avg_request_size_kb'] + params['avg_response_size_kb']) *
                          8 / 1024)
        api_bandwidth += bandwidth_mbps

    # Internal traffic
    internal_bandwidth = (
        self.bandwidth_factors['internal_traffic']['service_mesh']['requests_per_second'] *
        self.bandwidth_factors['internal_traffic']['service_mesh']['avg_payload_kb'] *
        8 / 1024 +
        self.bandwidth_factors['internal_traffic']['database_replication']['bandwidth_mbps'] +
        self.bandwidth_factors['internal_traffic']['monitoring']['metrics_bandwidth_mbps'] +
        self.bandwidth_factors['internal_traffic']['monitoring']['logs_bandwidth_mbps']
    )

    # CDN traffic
    cdn_bandwidth = (
        self.bandwidth_factors['cdn_traffic']['thumbnail_serving']['requests_per_minute'] *
        self.bandwidth_factors['cdn_traffic']['thumbnail_serving']['avg_size_kb'] *
        8 / 1024 / 60 +
        self.bandwidth_factors['cdn_traffic']['analytics_dashboard']['concurrent_users'] *
        self.bandwidth_factors['cdn_traffic']['analytics_dashboard']['bandwidth_per_user_mbps']
    )

    # Total with 50% headroom
    total_bandwidth = (upload_bandwidth + api_bandwidth +
                       internal_bandwidth + cdn_bandwidth) * 1.5

    return {
        'upload_bandwidth_mbps': upload_bandwidth,
        'api_bandwidth_mbps': api_bandwidth,
```

```
            'internal_bandwidth_mbps': internal_bandwidth,
            'cdn_bandwidth_mbps': cdn_bandwidth,
            'total_required_mbps': total_bandwidth,
            'recommended_connection': self._recommend_connection(total_bandwidth)
        }

    def _recommend_connection(self, bandwidth_mbps: float) -> str:
        """Recommend network connection based on bandwidth needs"""

        if bandwidth_mbps < 100:
            return "1 Gbps fiber (single connection)"
        elif bandwidth_mbps < 500:
            return "1 Gbps fiber (redundant connections)"
        elif bandwidth_mbps < 1000:
            return "10 Gbps fiber (single connection)"
        else:
            return "10 Gbps fiber (multiple connections with load balancing)"
```

Network Scaling Timeline

yaml

```yaml
network_scaling_timeline:
  phase_1_local:
    connection: "1 Gbps Fiber"
    bandwidth_allocation:
      upload: 200 Mbps
      download: 300 Mbps
      internal: 100 Mbps
    peak_utilization: 60%
    redundancy: "4G LTE backup"

  phase_2_hybrid:
    primary: "1 Gbps Fiber (Redundant)"
    secondary: "500 Mbps Cable"
    bandwidth_allocation:
      upload: 400 Mbps
      download: 600 Mbps
      cloud_interconnect: 200 Mbps
    peak_utilization: 70%
    redundancy: "Active-Active dual connection"

  phase_3_cloud:
    datacenter: "10 Gbps Direct Connect"
    internet: "1 Gbps redundant"
    cdn_integration: "CloudFront"
    bandwidth_allocation:
      inter_region: 500 Mbps
      internet_egress: 2 Gbps
      cdn_origin: 1 Gbps
    peak_utilization: 50%

  phase_4_global:
    regions:
      - location: "US-East"
        connection: "10 Gbps redundant"
        peering: "Major ISPs and CDNs"
      - location: "EU-West"
        connection: "10 Gbps redundant"
        peering: "Regional ISPs"
      - location: "APAC"
        connection: "10 Gbps redundant"
        peering: "Local CDNs"
    global_backbone: "100 Gbps private network"
    edge_locations: 25
```

Cost Projections by Component

Comprehensive Cost Model

python

```python
class CostProjectionModel:
    """Model infrastructure costs across all components"""

    def __init__(self):
        self.cost_factors = {
            'compute': {
                'local': {
                    'hardware_amortization': 500,  # $/month over 3 years
                    'electricity': 200,  # $/month
                    'cooling': 100,  # $/month
                    'maintenance': 100  # $/month
                },
                'cloud': {
                    'on_demand': {
                        'c6i.xlarge': 0.17,  # $/hour
                        'c6i.2xlarge': 0.34,
                        'c6i.4xlarge': 0.68,
                        'g5.xlarge': 1.006,
                        'g5.2xlarge': 1.212,
                        'g5.4xlarge': 1.624
                    },
                    'reserved_1year': 0.65,  # Multiplier for on-demand
                    'spot': 0.30  # Multiplier for on-demand
                }
            },
            'storage': {
                'local': {
                    'nvme_per_tb': 100,  # $/TB upfront
                    'ssd_per_tb': 50,
                    'hdd_per_tb': 25
                },
                'cloud': {
                    's3_standard': 0.023,  # $/GB/month
                    's3_ia': 0.0125,
                    's3_glacier': 0.004,
                    's3_deep_archive': 0.00099,
                    'ebs_gp3': 0.08,
                    'ebs_io2': 0.125
                }
            },
            'network': {
                'local': {
                    'fiber_1gbps': 500,  # $/month
                    'fiber_10gbps': 2000
                },
                'cloud': {
```

```
            'data_transfer_out': 0.09,  # $/GB
            'data_transfer_between_regions': 0.02,
            'direct_connect_port': 0.30,  # $/hour
            'nat_gateway': 0.045  # $/hour
        }
    },
    'services': {
        'youtube_api': 0,  # Free within quota
        'openai_api': {
            'gpt4_input': 0.01,  # $/1K tokens
            'gpt4_output': 0.03
        },
        'elevenlabs_tts': 0.30,  # $/1K characters
        'monitoring': {
            'datadog': 15,  # $/host/month
            'elasticsearch': 50  # $/month for small cluster
        }
    }
}

def project_monthly_costs(self, scale_parameters: dict) -> dict:
    """Project monthly costs based on scale parameters"""

    # Compute costs
    if scale_parameters['infrastructure'] == 'local':
        compute_cost = sum(self.cost_factors['compute']['local'].values())
    else:
        compute_cost = self._calculate_cloud_compute_cost(scale_parameters)

    # Storage costs
    storage_cost = self._calculate_storage_cost(scale_parameters)

    # Network costs
    network_cost = self._calculate_network_cost(scale_parameters)

    # Service costs
    service_cost = self._calculate_service_cost(scale_parameters)

    # Total with breakdown
    total_cost = compute_cost + storage_cost + network_cost + service_cost

    return {
        'total_monthly_cost': total_cost,
        'cost_breakdown': {
            'compute': compute_cost,
            'storage': storage_cost,
            'network': network_cost,
```

```
            'services': service_cost
        },
        'cost_per_video': total_cost / scale_parameters['videos_per_month'],
        'cost_per_channel': total_cost / scale_parameters['channels'],
        'optimization_opportunities': self._identify_cost_optimizations(
            scale_parameters,
            total_cost
        )
    }
```

Cost Projection Timeline

yaml

```yaml
cost_projection_timeline:
  year_1:
    month_1_3:
      infrastructure: "Local"
      monthly_cost: $500
      breakdown:
        hardware_amortization: $300
        utilities: $150
        internet: $50
      cost_per_video: $2.78

    month_4_6:
      infrastructure: "Hybrid"
      monthly_cost: $2,500
      breakdown:
        local_infrastructure: $500
        cloud_compute: $800
        cloud_storage: $400
        api_services: $600
        network: $200
      cost_per_video: $2.78

    month_7_12:
      infrastructure: "Cloud-Primary"
      monthly_cost: $8,000
      breakdown:
        compute_reserved: $3,000
        compute_spot: $1,000
        storage_tiered: $1,500
        network_cdn: $1,000
        api_services: $1,200
        monitoring: $300
      cost_per_video: $1.78

  year_2:
    average_monthly: $15,000
    infrastructure: "Multi-Region Cloud"
    breakdown:
      compute_global: $6,000
      storage_distributed: $3,000
      network_global: $2,500
      api_services: $2,500
      operations: $1,000
    cost_per_video: $1.67

  optimization_impact:
```

reserved_instances: -35%
spot_instances: -70%
storage_tiering: -60%
cdn_caching: -40%
total_optimization: -45%

ROI Analysis

python

```python
class ROIAnalyzer:
    """Analyze return on investment for infrastructure spending"""

    def __init__(self):
        self.revenue_model = {
            'revenue_per_video': 15,  # Average revenue per video
            'revenue_growth_rate': 1.15,  # 15% monthly growth
            'channel_multiplier': 1.2,  # Revenue multiplier per channel
        }

    def calculate_roi(self, cost_projections: dict, revenue_projections: dict) -> dict:
        """Calculate ROI metrics"""

        roi_metrics = []
        cumulative_investment = 0
        cumulative_revenue = 0

        for month in range(1, 25):  # 2 year projection
            # Get monthly costs and revenue
            monthly_cost = cost_projections[f'month_{month}']['total']
            monthly_revenue = revenue_projections[f'month_{month}']['total']

            cumulative_investment += monthly_cost
            cumulative_revenue += monthly_revenue

            monthly_profit = monthly_revenue - monthly_cost
            roi_percentage = ((cumulative_revenue - cumulative_investment) /
                              cumulative_investment * 100) if cumulative_investment > 0 else 0

            roi_metrics.append({
                'month': month,
                'monthly_cost': monthly_cost,
                'monthly_revenue': monthly_revenue,
                'monthly_profit': monthly_profit,
                'cumulative_roi': roi_percentage,
                'payback_achieved': cumulative_revenue >= cumulative_investment
            })

        # Find break-even point
        break_even_month = next((m['month'] for m in roi_metrics
                                 if m['payback_achieved']), None)

        return {
            'roi_timeline': roi_metrics,
            'break_even_month': break_even_month,
            'two_year_roi': roi_metrics[-1]['cumulative_roi'],
```

```
        'monthly_profit_year2': sum(m['monthly_profit']
                                    for m in roi_metrics[12:24]) / 12
    }
```

Scaling Timeline and Triggers

Automated Scaling Decision Framework

python

```python
class ScalingDecisionEngine:
    """Automated scaling decision engine based on metrics and triggers"""

    def __init__(self):
        self.scaling_triggers = {
            'compute_scaling': {
                'scale_up': {
                    'cpu_utilization': {'threshold': 70, 'duration': 300},
                    'memory_utilization': {'threshold': 80, 'duration': 300},
                    'queue_depth': {'threshold': 100, 'duration': 60},
                    'response_time_p95': {'threshold': 1000, 'duration': 300}
                },
                'scale_down': {
                    'cpu_utilization': {'threshold': 30, 'duration': 900},
                    'memory_utilization': {'threshold': 40, 'duration': 900},
                    'queue_depth': {'threshold': 10, 'duration': 600}
                }
            },
            'storage_scaling': {
                'add_capacity': {
                    'available_space_percent': {'threshold': 20, 'duration': 0},
                    'growth_rate_gb_day': {'threshold': 100, 'duration': 86400},
                    'iops_utilization': {'threshold': 80, 'duration': 600}
                },
                'tier_migration': {
                    'data_age_days': {'hot_to_warm': 30, 'warm_to_cold': 90},
                    'access_frequency': {'threshold': 0.01, 'duration': 604800}
                }
            },
            'infrastructure_migration': {
                'local_to_hybrid': {
                    'daily_videos': {'threshold': 20},
                    'compute_utilization': {'threshold': 80},
                    'storage_remaining_days': {'threshold': 30}
                },
                'hybrid_to_cloud': {
                    'daily_videos': {'threshold': 100},
                    'operational_complexity': {'threshold': 'high'},
                    'cost_efficiency': {'threshold': 1.2}  # Cloud becomes cheaper
                }
            }
        }

    async def evaluate_scaling_decision(self, current_metrics: dict) -> dict:
        """Evaluate current metrics and make scaling decisions"""
```

```
        decisions = []

        # Check compute scaling triggers
        for trigger_type, triggers in self.scaling_triggers['compute_scaling'].items():
            for metric, criteria in triggers.items():
                if self._check_trigger(current_metrics, metric, criteria):
                    decisions.append({
                        'type': 'compute',
                        'action': trigger_type,
                        'metric': metric,
                        'current_value': current_metrics[metric],
                        'threshold': criteria['threshold']
                    })

        # Check storage scaling triggers
        for trigger_type, triggers in self.scaling_triggers['storage_scaling'].items():
            for metric, criteria in triggers.items():
                if self._check_trigger(current_metrics, metric, criteria):
                    decisions.append({
                        'type': 'storage',
                        'action': trigger_type,
                        'metric': metric,
                        'current_value': current_metrics.get(metric),
                        'threshold': criteria
                    })

        # Check infrastructure migration triggers
        migration_score = self._calculate_migration_score(current_metrics)
        if migration_score > 0.7:
            decisions.append({
                'type': 'infrastructure',
                'action': 'migrate',
                'from': current_metrics['current_infrastructure'],
                'to': self._determine_target_infrastructure(current_metrics),
                'migration_score': migration_score
            })

        return {
            'timestamp': datetime.utcnow(),
            'decisions': decisions,
            'recommended_actions': self._prioritize_decisions(decisions),
            'estimated_impact': self._estimate_scaling_impact(decisions)
        }
```

Scaling Timeline Visualization

yaml

```yaml
scaling_timeline:
  month_1:
    triggers_monitored:
      - daily_video_count
      - cpu_utilization
      - storage_growth_rate
    thresholds:
      daily_videos: 10
      cpu_average: 60%
      storage_days_remaining: 60
    expected_action: "Monitor only"

  month_2:
    triggers_monitored:
      - all_previous
      - api_rate_limits
      - concurrent_workflows
    thresholds:
      daily_videos: 15
      cpu_peak: 80%
      storage_days_remaining: 45
    expected_action: "Prepare hybrid infrastructure"

  month_3:
    triggers_monitored:
      - all_previous
      - cost_per_video
      - operational_complexity
    thresholds:
      daily_videos: 20
      cpu_sustained: 75%
      storage_days_remaining: 30
    expected_action: "Initiate hybrid deployment"

  month_4_6:
    infrastructure: "Hybrid"
    new_triggers:
      - cloud_burst_frequency
      - multi_region_latency
      - cost_optimization_score
    thresholds:
      daily_videos: 50
      cloud_burst_daily: 5
      cost_efficiency: 0.8
    expected_action: "Optimize hybrid balance"
```

```yaml
  month_7_9:
    triggers_monitored:
      - global_latency
      - region_specific_demand
      - compliance_requirements
    thresholds:
      daily_videos: 100
      global_users: 10000
      latency_p95: 200ms
    expected_action: "Plan cloud migration"

  month_10_12:
    infrastructure: "Cloud-native"
    triggers:
      - multi_region_demand
      - edge_cache_hit_rate
      - global_availability
    thresholds:
      daily_videos: 150
      regions_active: 3
      availability_target: 99.95%
    expected_action: "Execute cloud migration"

  year_2:
    infrastructure: "Global multi-region"
    advanced_triggers:
      - predictive_demand
      - cost_anomalies
      - competitive_pressure
    automation_level: "Fully autonomous scaling"
```

Performance Models

Load Testing Specifications

Comprehensive Load Testing Framework

python

```python
class LoadTestingFramework:
    """Define and execute load testing scenarios"""

    def __init__(self):
        self.test_scenarios = {
            'baseline': {
                'description': 'Normal daily operations',
                'duration': 3600,  # 1 hour
                'ramp_up': 300,  # 5 minutes
                'users': {
                    'concurrent': 100,
                    'arrival_rate': '10/s'
                },
                'workload_mix': {
                    'api_requests': 0.4,
                    'video_generation': 0.3,
                    'analytics_queries': 0.2,
                    'admin_operations': 0.1
                }
            },
            'peak_load': {
                'description': 'Peak hours simulation',
                'duration': 1800,  # 30 minutes
                'ramp_up': 600,  # 10 minutes
                'users': {
                    'concurrent': 500,
                    'arrival_rate': '50/s'
                },
                'workload_mix': {
                    'api_requests': 0.3,
                    'video_generation': 0.5,
                    'analytics_queries': 0.15,
                    'admin_operations': 0.05
                }
            },
            'sustained_load': {
                'description': '24-hour sustained load',
                'duration': 86400,  # 24 hours
                'ramp_up': 1800,  # 30 minutes
                'users': {
                    'concurrent': 300,
                    'arrival_rate': '30/s'
                },
                'workload_mix': {
                    'api_requests': 0.35,
                    'video_generation': 0.4,
```

```
            'analytics_queries': 0.2,
            'admin_operations': 0.05
        }
    }
}

self.performance_targets = {
    'response_time': {
        'api_requests': {'p50': 100, 'p95': 500, 'p99': 1000},  # ms
        'video_generation': {'p50': 30000, 'p95': 60000, 'p99': 120000},
        'analytics_queries': {'p50': 200, 'p95': 1000, 'p99': 2000}
    },
    'throughput': {
        'api_requests': 1000,  # requests/second
        'video_generation': 10,  # videos/minute
        'analytics_queries': 100  # queries/second
    },
    'error_rate': {
        'target': 0.001,  # 0.1%
        'critical': 0.01  # 1%
    }
}
```

K6 Load Test Implementation

javascript

```
// load-test.js - K6 Load Testing Script
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Rate, Trend } from 'k6/metrics';
import { randomItem } from 'https://jslib.k6.io/k6-utils/1.2.0/index.js';

// Custom metrics
const errorRate = new Rate('errors');
const apiLatency = new Trend('api_latency');
const videoGenLatency = new Trend('video_generation_latency');

// Test configuration
export const options = {
  scenarios: {
    baseline_load: {
      executor: 'ramping-arrival-rate',
      startRate: 10,
      timeUnit: '1s',
      preAllocatedVUs: 100,
      maxVUs: 500,
      stages: [
        { duration: '5m', target: 10 },   // Ramp up
        { duration: '50m', target: 30 },  // Stay at 30 RPS
        { duration: '5m', target: 0 },    // Ramp down
      ],
    },
    spike_test: {
      executor: 'ramping-vus',
      startVUs: 0,
      stages: [
        { duration: '2m', target: 100 },
        { duration: '5m', target: 100 },
        { duration: '2m', target: 1000 }, // Spike to 1000 users
        { duration: '5m', target: 1000 },
        { duration: '2m', target: 100 },
        { duration: '5m', target: 100 },
        { duration: '2m', target: 0 },
      ],
    },
  },
  thresholds: {
    'http_req_duration{type:api}': ['p(95)<500', 'p(99)<1000'],
    'http_req_duration{type:video}': ['p(95)<60000', 'p(99)<120000'],
    'errors': ['rate<0.01'],
    'http_req_failed': ['rate<0.01'],
  },
```

```
};

const BASE_URL = 'https://api.ytempire.com';

// Test scenarios
const scenarios = {
  apiRequest: () => {
    const endpoints = ['/trends', '/channels', '/videos', '/analytics'];
    const endpoint = randomItem(endpoints);

    const params = {
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ${__ENV.API_TOKEN}',
      },
      tags: { type: 'api' },
    };

    const res = http.get(`${BASE_URL}${endpoint}`, params);

    check(res, {
      'API status is 200': (r) => r.status === 200,
      'API response time < 500ms': (r) => r.timings.duration < 500,
    });

    errorRate.add(res.status !== 200);
    apiLatency.add(res.timings.duration);
  },

  videoGeneration: () => {
    const payload = JSON.stringify({
      channel_id: 'test_channel_' + Math.floor(Math.random() * 10),
      topic: 'Test Topic ' + Date.now(),
      style: randomItem(['educational', 'entertainment', 'tutorial']),
    });

    const params = {
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ${__ENV.API_TOKEN}',
      },
      tags: { type: 'video' },
      timeout: '120s',
    };

    const res = http.post(`${BASE_URL}/videos/generate`, payload, params);
```

```
    check(res, {
      'Video generation accepted': (r) => r.status === 202,
      'Video generation time < 60s': (r) => r.timings.duration < 60000,
    });

    errorRate.add(res.status !== 202);
    videoGenLatency.add(res.timings.duration);
  },
};

export default function() {
  const weights = {
    apiRequest: 0.7,
    videoGeneration: 0.3,
  };

  const rand = Math.random();
  if (rand < weights.apiRequest) {
    scenarios.apiRequest();
  } else {
    scenarios.videoGeneration();
  }

  sleep(1);
}

export function handleSummary(data) {
  return {
    'load-test-results.json': JSON.stringify(data),
    'load-test-report.html': htmlReport(data),
  };
}
```

Performance Baseline Establishment

python

```python
class PerformanceBaseline:
    """Establish and track performance baselines"""

    def __init__(self):
        self.baseline_metrics = {}
        self.degradation_thresholds = {
            'response_time': 1.2,  # 20% degradation
            'throughput': 0.8,     # 20% reduction
            'error_rate': 2.0,     # 2x increase
            'cpu_utilization': 1.3,  # 30% increase
        }

    async def establish_baseline(self, test_results: dict) -> dict:
        """Establish performance baseline from test results"""

        baseline = {
            'timestamp': datetime.utcnow(),
            'configuration': test_results['configuration'],
            'metrics': {
                'response_times': {
                    'p50': test_results['percentiles']['50'],
                    'p95': test_results['percentiles']['95'],
                    'p99': test_results['percentiles']['99'],
                },
                'throughput': {
                    'requests_per_second': test_results['rps'],
                    'successful_requests': test_results['success_rate'],
                },
                'resource_utilization': {
                    'cpu_average': test_results['cpu_avg'],
                    'memory_average': test_results['memory_avg'],
                    'gpu_average': test_results.get('gpu_avg', 0),
                },
                'error_rates': {
                    'total_errors': test_results['error_rate'],
                    'timeout_errors': test_results['timeout_rate'],
                    'server_errors': test_results['5xx_rate'],
                }
            }
        }

        self.baseline_metrics[test_results['scenario']] = baseline
        return baseline

    def detect_performance_regression(self, current_metrics: dict, scenario: str) -> dict:
        """Detect performance regression from baseline"""
```

```python
        baseline = self.baseline_metrics.get(scenario)
        if not baseline:
            return {'regression_detected': False, 'reason': 'No baseline established'}

        regressions = []

        # Check response time regression
        for percentile in ['p50', 'p95', 'p99']:
            current = current_metrics['response_times'][percentile]
            baseline_value = baseline['metrics']['response_times'][percentile]

            if current > baseline_value * self.degradation_thresholds['response_time']:
                regressions.append({
                    'metric': f'response_time_{percentile}',
                    'baseline': baseline_value,
                    'current': current,
                    'degradation': (current - baseline_value) / baseline_value * 100
                })

        # Check throughput regression
        current_throughput = current_metrics['throughput']['requests_per_second']
        baseline_throughput = baseline['metrics']['throughput']['requests_per_second']

        if current_throughput < baseline_throughput * self.degradation_thresholds['throughput']:
            regressions.append({
                'metric': 'throughput',
                'baseline': baseline_throughput,
                'current': current_throughput,
                'degradation': (baseline_throughput - current_throughput) / baseline_throughput * 100
            })

        return {
            'regression_detected': len(regressions) > 0,
            'regressions': regressions,
            'severity': self._calculate_severity(regressions)
        }
```

Stress Testing Scenarios

Extreme Load Scenarios

python

```python
class StressTestScenarios:
    """Define stress testing scenarios to find breaking points"""

    def __init__(self):
        self.stress_scenarios = {
            'traffic_spike': {
                'description': 'Sudden 10x traffic spike',
                'implementation': {
                    'normal_load': 100,  # users
                    'spike_load': 1000,  # users
                    'spike_duration': 600,  # 10 minutes
                    'recovery_time': 900  # 15 minutes
                },
                'expected_behavior': {
                    'auto_scaling': True,
                    'degraded_mode': 'possible',
                    'circuit_breakers': 'may_activate',
                    'recovery': 'automatic'
                }
            },
            'resource_exhaustion': {
                'description': 'Gradual resource exhaustion',
                'implementation': {
                    'starting_load': 50,
                    'increment': 10,  # users per minute
                    'target': 'system_failure',
                    'metrics_watched': ['cpu', 'memory', 'connections']
                },
                'expected_behavior': {
                    'warning_at': '80% utilization',
                    'throttling_at': '90% utilization',
                    'failure_mode': 'graceful_degradation'
                }
            },
            'cascade_failure': {
                'description': 'Service dependency failure cascade',
                'implementation': {
                    'failed_service': 'database',
                    'failure_duration': 300,  # 5 minutes
                    'dependent_services': ['api', 'analytics', 'publisher'],
                    'recovery_order': ['database', 'api', 'analytics', 'publisher']
                },
                'expected_behavior': {
                    'circuit_breakers': 'must_activate',
                    'fallback_mode': 'cache_only',
                    'data_consistency': 'eventual'
```

```
            }
        },
        'sustained_overload': {
            'description': '24-hour sustained 2x capacity',
            'implementation': {
                'load_multiplier': 2.0,
                'duration': 86400,  # 24 hours
                'checkpoints': [1, 6, 12, 24],  # hours
            },
            'expected_behavior': {
                'auto_scaling': 'continuous',
                'cost_impact': '3x normal',
                'performance_target': 'maintain_slo'
            }
        }
    }

def generate_chaos_experiments(self) -> list:
    """Generate chaos engineering experiments"""

    experiments = [
        {
            'name': 'Random Pod Failure',
            'target': 'kubernetes_pods',
            'action': 'terminate_random',
            'frequency': '1 per hour',
            'duration': '8 hours',
            'expected_recovery': '< 60 seconds'
        },
        {
            'name': 'Network Partition',
            'target': 'availability_zones',
            'action': 'block_traffic',
            'duration': '5 minutes',
            'expected_behavior': 'failover_to_healthy_az'
        },
        {
            'name': 'API Rate Limit Hit',
            'target': 'youtube_api',
            'action': 'exhaust_quota',
            'expected_behavior': 'graceful_degradation'
        },
        {
            'name': 'Storage Failure',
            'target': 's3_bucket',
            'action': 'make_unavailable',
            'duration': '10 minutes',
```

```
            'expected_behavior': 'fallback_to_secondary'
        }
    ]

    return experiments
```

Stress Test Execution Framework

yaml

```yaml
stress_test_execution:
  preparation:
    - backup_current_state
    - notify_stakeholders
    - enable_enhanced_monitoring
    - prepare_rollback_plan

  execution_phases:
    phase_1_baseline:
      duration: 30_minutes
      load: normal
      purpose: establish_baseline

    phase_2_ramp_up:
      duration: 15_minutes
      load: gradually_increase
      target: 2x_normal
      monitoring:
        - response_times
        - error_rates
        - resource_utilization

    phase_3_stress:
      duration: 60_minutes
      load: stress_level
      variations:
        - sustained_high_load
        - spike_patterns
        - wave_patterns
      break_points:
        - first_errors
        - degraded_performance
        - system_failure

    phase_4_recovery:
      duration: 30_minutes
      load: gradually_decrease
      monitoring:
        - recovery_time
        - data_consistency
        - system_stability

  analysis:
    metrics_collected:
      - maximum_sustainable_load
      - breaking_points
```

```
    - recovery_characteristics
    - bottleneck_identification

  recommendations:
    - capacity_adjustments
    - architecture_improvements
    - monitoring_enhancements
    - runbook_updates
```

Capacity Planning Formulas

Mathematical Models for Capacity Planning

python

```python
class CapacityPlanningFormulas:
    """Mathematical formulas for capacity planning"""

    @staticmethod
    def little_law_calculation(arrival_rate: float, response_time: float) -> float:
        """
        Little's Law: L = λ * W
        L = average number of requests in system
        λ = arrival rate
        W = average response time
        """
        return arrival_rate * response_time

    @staticmethod
    def queuing_theory_mm1(arrival_rate: float, service_rate: float) -> dict:
        """
        M/M/1 Queue calculations
        """
        if arrival_rate >= service_rate:
            return {'error': 'Arrival rate must be less than service rate'}

        utilization = arrival_rate / service_rate
        average_queue_length = utilization / (1 - utilization)
        average_wait_time = average_queue_length / arrival_rate
        average_system_time = 1 / (service_rate - arrival_rate)

        return {
            'utilization': utilization,
            'average_queue_length': average_queue_length,
            'average_wait_time': average_wait_time,
            'average_system_time': average_system_time
        }

    @staticmethod
    def amdahl_law(parallel_fraction: float, num_processors: int) -> float:
        """
        Amdahl's Law for parallel speedup
        Speedup = 1 / ((1 - P) + P/N)
        P = parallel fraction
        N = number of processors
        """
        serial_fraction = 1 - parallel_fraction
        speedup = 1 / (serial_fraction + parallel_fraction / num_processors)
        return speedup

    @staticmethod
```

```python
    def universal_scalability_law(
        alpha: float,  # contention
        beta: float,   # coherency
        num_processors: int
    ) -> float:
        """
        USL: C(N) = N / (1 + α(N-1) + βN(N-1))
        """
        capacity = num_processors / (1 + alpha * (num_processors - 1) +
                                     beta * num_processors * (num_processors - 1))
        return capacity

    @staticmethod
    def storage_growth_projection(
        current_size: float,
        growth_rate: float,
        time_periods: int,
        compression_ratio: float = 0.5
    ) -> dict:
        """
        Project storage growth with compression
        """
        projections = []
        size = current_size

        for period in range(1, time_periods + 1):
            size = size * (1 + growth_rate)
            compressed_size = size * compression_ratio

            projections.append({
                'period': period,
                'raw_size': size,
                'compressed_size': compressed_size,
                'total_with_replicas': compressed_size * 3  # 3x replication
            })

        return {
            'projections': projections,
            'total_growth': size / current_size,
            'average_growth_rate': (size / current_size) ** (1/time_periods) - 1
        }
```

Workload Modeling

```python
```

```python
class WorkloadModel:
    """Model different workload patterns"""

    def __init__(self):
        self.workload_patterns = {
            'steady_state': self.steady_state_model,
            'daily_cycle': self.daily_cycle_model,
            'growth_trend': self.growth_trend_model,
            'seasonal': self.seasonal_model,
            'viral_spike': self.viral_spike_model
        }

    def steady_state_model(self, base_load: float, duration: int) -> list:
        """Constant workload"""
        return [base_load] * duration

    def daily_cycle_model(self, base_load: float, duration_days: int) -> list:
        """Daily cyclical pattern"""
        hourly_multipliers = [
            0.3, 0.2, 0.2, 0.3, 0.5, 0.7,  # 00:00 - 05:00
            1.0, 1.2, 1.5, 1.8, 2.0, 2.2,  # 06:00 - 11:00
            2.0, 1.8, 1.6, 1.4, 1.5, 1.8,  # 12:00 - 17:00
            2.0, 2.2, 2.0, 1.5, 1.0, 0.5   # 18:00 - 23:00
        ]

        workload = []
        for day in range(duration_days):
            for hour, multiplier in enumerate(hourly_multipliers):
                workload.append(base_load * multiplier)

        return workload

    def growth_trend_model(
        self,
        initial_load: float,
        growth_rate: float,
        duration: int
    ) -> list:
        """Exponential growth pattern"""
        workload = []
        current_load = initial_load

        for period in range(duration):
            workload.append(current_load)
            current_load *= (1 + growth_rate)
```

```
        return workload

    def viral_spike_model(
        self,
        base_load: float,
        spike_multiplier: float,
        spike_duration: int,
        total_duration: int
    ) -> list:
        """Model viral content spike"""
        workload = []
        spike_start = total_duration // 3  # Spike in middle third

        for period in range(total_duration):
            if spike_start <= period < spike_start + spike_duration:
                # Gaussian spike
                spike_position = (period - spike_start) / spike_duration
                spike_value = spike_multiplier * np.exp(-((spike_position - 0.5) ** 2) / 0.1)
                workload.append(base_load * (1 + spike_value))
            else:
                workload.append(base_load)

        return workload

    def combine_models(self, models: list, weights: list) -> list:
        """Combine multiple workload models"""
        combined = []

        for i in range(len(models[0])):
            value = sum(model[i] * weight
                        for model, weight in zip(models, weights))
            combined.append(value)

        return combined
```

Resource Optimization Strategies

Multi-Dimensional Optimization

python

```python
class ResourceOptimizer:
    """Optimize resources across multiple dimensions"""

    def __init__(self):
        self.optimization_dimensions = {
            'cost': CostOptimizer(),
            'performance': PerformanceOptimizer(),
            'reliability': ReliabilityOptimizer(),
            'scalability': ScalabilityOptimizer()
        }

        self.constraints = {
            'budget': 50000,  # Monthly budget
            'slo_target': 0.999,  # 99.9% availability
            'latency_target': 200,  # ms p95
            'min_headroom': 0.3  # 30% capacity headroom
        }

    async def optimize_resources(self, current_state: dict) -> dict:
        """Find optimal resource configuration"""

        # Define optimization problem
        problem = {
            'variables': {
                'compute_instances': {'min': 3, 'max': 50},
                'instance_types': ['c6i.large', 'c6i.xlarge', 'c6i.2xlarge'],
                'storage_tiers': {'hot': (0, 1), 'warm': (0, 1), 'cold': (0, 1)},
                'cache_size': {'min': 0, 'max': 1000},  # GB
                'cdn_locations': {'min': 0, 'max': 25}
            },
            'objectives': {
                'minimize_cost': 0.4,
                'maximize_performance': 0.3,
                'maximize_reliability': 0.2,
                'maximize_scalability': 0.1
            }
        }

        # Generate candidate solutions
        candidates = self._generate_candidates(problem, num_candidates=1000)

        # Evaluate each candidate
        evaluated_solutions = []
        for candidate in candidates:
            evaluation = await self._evaluate_solution(candidate, current_state)
```

```
        if self._meets_constraints(evaluation):
            evaluated_solutions.append({
                'solution': candidate,
                'evaluation': evaluation,
                'score': self._calculate_weighted_score(evaluation, problem['objectives'])
            })

    # Select best solution
    best_solution = max(evaluated_solutions, key=lambda x: x['score'])

    # Generate implementation plan
    implementation_plan = self._create_implementation_plan(
        current_state,
        best_solution['solution']
    )

    return {
        'optimal_configuration': best_solution['solution'],
        'expected_metrics': best_solution['evaluation'],
        'improvement_over_current': self._calculate_improvement(
            current_state,
            best_solution['evaluation']
        ),
        'implementation_plan': implementation_plan,
        'roi_analysis': self._calculate_optimization_roi(
            current_state,
            best_solution
        )
    }
```