Remote Integrated Services (RIS) - Low Level Design

Detailed technical implementation specifications for the RIS distributed system

Introduction

This Low-Level Design document provides comprehensive technical specifications for the Remote Integrated Services (RIS) System implementation. It includes detailed class structures, method signatures, database schemas, API endpoints, protocol specifications, and implementation specifics for both server and client components.

Design Principles

⚡ Simplicity

Clear, readable, and maintainable code structures with comprehensive documentation and standardized patterns.

🚀 Performance

Optimized for high throughput and low latency with efficient algorithms and resource management.

🔒 Security

Security-by-design with comprehensive protection mechanisms and defense-in-depth strategies.

📈 Scalability

Horizontal and vertical scaling capabilities with load balancing and distributed processing.

RIS Server Component Design

Main Application Structure

```
// App represents the main application structure
type App struct {
    Router              *mux.Router                 // HTTP router fo
    AccessTokenHashMap  map[string]int64            // JWT token expi
    Socket              *websocket.Conn             // WebSocket conn
    SocketClients       []*websocket.Conn           // Multiple WebSo
    DBClient            *mongo.Client               // MongoDB client
    RedisClient         *redis.Client               // Redis client fo
    Config              *Config                     // Application con
    Logger              *logrus.Logger              // Structured logg
    TaskManager         *TaskManager                // Task execution
    ClientManager       *ClientManager              // Connected clien
    SecurityManager     *SecurityManager            // Security and au
}

// Configuration structure for the application
type Config struct {
    Server struct {
        Host            string        `json:"host"`
        Port            int           `json:"port"`
        TLSEnabled      bool          `json:"tls_enabled"`
        CertFile        string        `json:"cert_file"`
        KeyFile         string        `json:"key_file"`
        ReadTimeout     time.Duration `json:"read_timeout"`
        WriteTimeout    time.Duration `json:"write_timeout"`
    } `json:"server"`

    Database struct {
        MongoURI        string `json:"mongo_uri"`
        DatabaseName    string `json:"database_name"`
        MaxPoolSize     uint64 `json:"max_pool_size"`
        ConnectionTimeout time.Duration `json:"connection_timeout"`
    } `json:"database"`

    Redis struct {
        URI             string `json:"uri"`
        Password        string `json:"password"`
        Database        int    `json:"database"`
        PoolSize        int    `json:"pool_size"`
```

```
    } `json:"redis"`

    Security struct {
        JWTSecret          string        `json:"jwt_secret"`
        JWTExpiry          time.Duration `json:"jwt_expiry"`
        RateLimitEnabled   bool          `json:"rate_limit_enabled"`
        MaxRequestsPerMin  int           `json:"max_requests_per_min"`
    } `json:"security"`
}
```

Client Management System

Copy

```
// ClientManager handles all connected RIS clients
type ClientManager struct {
    clients           map[string]*ConnectedClient
    clientsMutex      sync.RWMutex
    connectionPool    *ConnectionPool
    heartbeatInterval time.Duration
    logger            *logrus.Logger
}

// ConnectedClient represents a connected RIS client
type ConnectedClient struct {
    ID                string                 `json:"id"`
    Name              string                 `json:"name"`
    Host              string                 `json:"host"`
    Connection        *websocket.Conn        `json:"-"`
    Status            string                 `json:"status"`
    LastSeen          time.Time              `json:"last_seen"`
    Capabilities      []string               `json:"capabilities"`
    Version           string                 `json:"version"`
    TenantCode        string                 `json:"tenant_code"`
    MessageQueue      chan *Message          `json:"-"`
    IsAuthenticated   bool                   `json:"is_authenticated"`
    SecurityContext   *SecurityContext       `json:"-"`
}

// Message structure for WebSocket communication
type Message struct {
    Type              string                 `json:"type"`
    ID                string                 `json:"id"`
    ClientID          string                 `json:"client_id"`
    Payload           interface{}            `json:"payload"`
```

```
    Timestamp     time.Time          `json:"timestamp"`
    Security      *SecurityInfo      `json:"security,omitempty"`
    Priority      int                `json:"priority"`
    TTL           time.Duration      `json:"ttl"`
}
```

Task Management System

```
// TaskManager handles task execution and orchestration
type TaskManager struct {
    taskQueue         chan *Task
    runningTasks      map[string]*Task
    taskHistory       *TaskHistory
    maxConcurrentTasks int
    logger            *logrus.Logger
    mutex             sync.RWMutex
}

// Task represents a task to be executed on a client
type Task struct {
    ID                string                 `bson:"_id" json:"id"`
    Type              string                 `bson:"type" json:"type"`
    ClientID          string                 `bson:"client_id" json:"client
    Status            string                 `bson:"status" json:"status"`
    Priority          int                    `bson:"priority" json:"priority
    CreatedAt         time.Time              `bson:"created_at" json:"create
    StartedAt         *time.Time             `bson:"started_at,omitempty" js
    CompletedAt       *time.Time             `bson:"completed_at,omitempty"
    Timeout           time.Duration          `bson:"timeout" json:"timeout"`
    RetryCount        int                    `bson:"retry_count" json:"retry
    MaxRetries        int                    `bson:"max_retries" json:"max_r
    Payload           TaskPayload            `bson:"payload" json:"payload"`
    Result            *TaskResult            `bson:"result,omitempty" json:"
    TenantCode        string                 `bson:"tenant_code" json:"tenan
}

// TaskPayload contains task-specific data
type TaskPayload struct {
    TaskType          string                 `bson:"task_type" json:"task_t
    Parameters        map[string]interface{} `bson:"parameters" json:"param
    Environment       map[string]string      `bson:"environment,omitempty"
    Dependencies      []string               `bson:"dependencies,omitempty"
}
```

```
// TaskResult contains task execution results
type TaskResult struct {
    Success         bool                `bson:"success" json:"success"
    Output          string              `bson:"output,omitempty" json:
    Error           string              `bson:"error,omitempty" json:"
    ExitCode        int                 `bson:"exit_code,omitempty" jso
    ExecutionTime   time.Duration       `bson:"execution_time" json:"ex
    ResourceUsage   *ResourceUsage      `bson:"resource_usage,omitempty
}
```

Database Design and Schema

MongoDB Collections

RIS Clients Collection

Copy

```
{
  "_id": ObjectId("..."),
  "unique_client_id": "uuid-string",
  "name": "client-production-001",
  "description": "Production environment client",
  "host": "192.168.1.100",
  "status": "active",
  "connection_status": "connected",
  "version": "1.0.0",
  "tenant_code": "acme-corp",
  "capabilities": ["docker_management", "system_monitoring", "file_operatio
  "security": {
    "certificate_fingerprint": "sha256:abc123...",
    "last_auth": ISODate("2025-10-07T10:30:00Z"),
    "auth_method": "certificate"
  },
  "configuration": {
    "heartbeat_interval": 30,
    "max_concurrent_tasks": 10,
    "allowed_operations": ["start", "stop", "restart", "status"]
  },
  "created_at": ISODate("2025-10-07T08:00:00Z"),
  "updated_at": ISODate("2025-10-07T10:30:00Z"),
  "last_seen": ISODate("2025-10-07T10:30:00Z")
}
```

Tasks Collection

Copy

```
{
  "_id": ObjectId("..."),
  "task_id": "task-uuid-string",
  "client_id": "client-uuid-string",
  "tenant_code": "acme-corp",
  "type": "docker_management",
```

```
  "status": "completed",
  "priority": 1,
  "payload": {
    "action": "start_service",
    "service_name": "webapp",
    "image": "nginx:latest",
    "ports": ["80:8080"],
    "environment": {
      "ENV": "production",
      "LOG_LEVEL": "info"
    }
  },
  "result": {
    "success": true,
    "output": "Service started successfully",
    "exit_code": 0,
    "execution_time_ms": 2500,
    "resource_usage": {
      "cpu_percent": 15.5,
      "memory_mb": 128,
      "disk_io_mb": 50
    }
  },
  "created_at": ISODate("2025-10-07T10:25:00Z"),
  "started_at": ISODate("2025-10-07T10:25:05Z"),
  "completed_at": ISODate("2025-10-07T10:25:07Z"),
  "timeout_seconds": 300,
  "retry_count": 0,
  "max_retries": 3
}
```

Audit Logs Collection

Copy

```
{
  "_id": ObjectId("..."),
  "timestamp": ISODate("2025-10-07T10:30:00Z"),
  "event_type": "task_execution",
  "action": "start_docker_service",
  "actor": {
    "type": "user",
    "id": "user-123",
    "email": "admin@acme-corp.com",
    "role": "admin"
  },
```

```
  "target": {
    "type": "client",
    "id": "client-uuid-string",
    "name": "client-production-001"
  },
  "tenant_code": "acme-corp",
  "result": "success",
  "details": {
    "task_id": "task-uuid-string",
    "service_name": "webapp",
    "execution_time_ms": 2500
  },
  "ip_address": "192.168.1.50",
  "user_agent": "RIS-Dashboard/1.0.0",
  "session_id": "session-uuid-string"
}
```

Database Indexes

Copy

```
// RIS Clients Collection Indexes
db.ris_clients.createIndex({"unique_client_id": 1}, {"unique": true})
db.ris_clients.createIndex({"tenant_code": 1, "status": 1})
db.ris_clients.createIndex({"connection_status": 1, "last_seen": 1})
db.ris_clients.createIndex({"host": 1})

// Tasks Collection Indexes
db.tasks.createIndex({"task_id": 1}, {"unique": true})
db.tasks.createIndex({"client_id": 1, "status": 1})
db.tasks.createIndex({"tenant_code": 1, "created_at": -1})
db.tasks.createIndex({"status": 1, "priority": -1})
db.tasks.createIndex({"created_at": 1}, {"expireAfterSeconds": 2592000}) //

// Audit Logs Collection Indexes
db.audit_logs.createIndex({"timestamp": -1})
db.audit_logs.createIndex({"tenant_code": 1, "timestamp": -1})
db.audit_logs.createIndex({"event_type": 1, "timestamp": -1})
db.audit_logs.createIndex({"actor.id": 1, "timestamp": -1})
db.audit_logs.createIndex({"timestamp": 1}, {"expireAfterSeconds": 7776000}
```

API Specifications and Protocols

REST API Endpoints

Authentication Endpoints

METHOD	ENDPOINT	DESCRIPTION	AUTHENTICATION
`POST`	`/auth/login`	User authentication and JWT token generation	Basic Auth
`POST`	`/auth/refresh`	Refresh JWT access token	Refresh Token
`POST`	`/auth/logout`	Logout and invalidate tokens	JWT
`GET`	`/auth/profile`	Get current user profile information	JWT

Client Management Endpoints

METHOD	ENDPOINT	DESCRIPTION	AUTHENTICATION
`GET`	`/platform/v1/ris`	List all RIS clients with filtering and pagination	JWT
`GET`	`/platform/v1/ris/{id}`	Get specific RIS client details	JWT
`POST`	`/platform/v1/ris`	Register new RIS client	JWT
`PUT`	`/platform/v1/ris/{id}`	Update RIS client configuration	JWT
`DELETE`	`/platform/v1/ris/{id}`	Remove RIS client from system	JWT

Task Management Endpoints

METHOD	ENDPOINT	DESCRIPTION	AUTHENTICATION
`POST`	`/platform/v1/task/execute`	Execute task on specified client	JWT

METHOD	ENDPOINT	DESCRIPTION	AUTHENTICATION
`GET`	`/platform/v1/task/{id}`	Get task status and results	JWT
`GET`	`/platform/v1/tasks`	List tasks with filtering and pagination	JWT
`DELETE`	`/platform/v1/task/{id}`	Cancel or abort running task	JWT

Task Execution Request Format

Copy

```
POST /platform/v1/task/execute
Content-Type: application/json
Authorization: Bearer <jwt_token>

{
  "client_id": "client-uuid-string",
  "task": {
    "type": "docker_management",
    "action": "start_service",
    "parameters": {
      "service_name": "webapp",
      "image": "nginx:latest",
      "ports": ["80:8080"],
      "environment": {
        "ENV": "production",
        "LOG_LEVEL": "info"
      },
      "volumes": ["/data:/app/data"],
      "restart_policy": "unless-stopped"
    },
    "timeout": 300,
    "max_retries": 3
  },
  "priority": 1,
  "async": false,
  "callback_url": "https://api.example.com/webhook/task-complete"
}
```

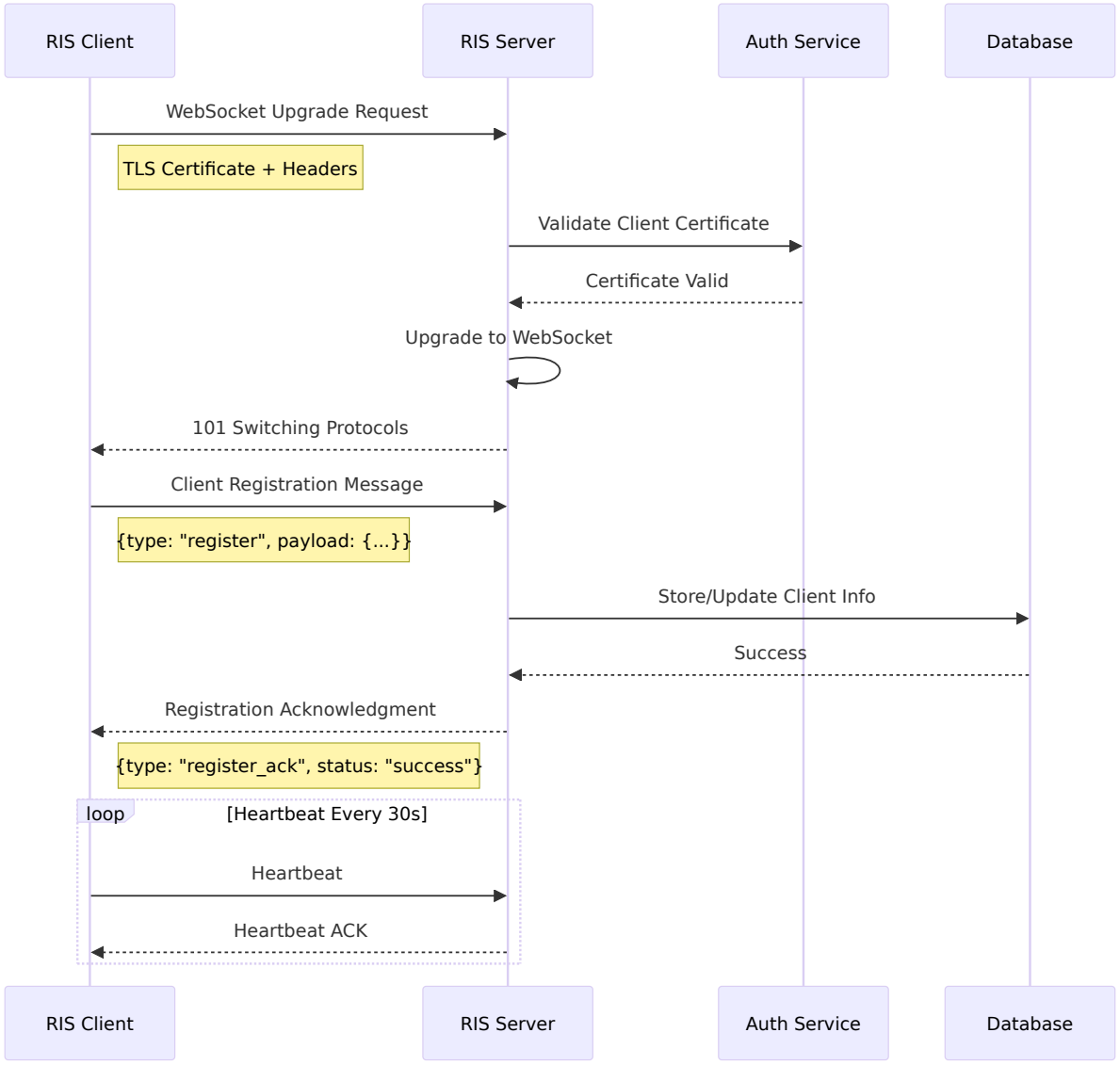
Task Execution Response Format

Copy

```
{
  "success": true,
  "data": {
    "task_id": "task-uuid-string",
    "status": "running",
    "created_at": "2025-10-07T10:30:00Z",
    "estimated_completion": "2025-10-07T10:35:00Z",
    "client": {
      "id": "client-uuid-string",
      "name": "client-production-001",
      "host": "192.168.1.100"
    }
  },
  "links": {
    "status": "/platform/v1/task/task-uuid-string",
    "logs": "/platform/v1/task/task-uuid-string/logs",
    "cancel": "/platform/v1/task/task-uuid-string"
  }
}
```

WebSocket Protocol Implementation

Connection Establishment



Message Types and Formats

Client Registration Message

Copy

```
{
  "type": "register",
  "timestamp": "2025-10-07T10:30:00Z",
  "payload": {
    "client_id": "uuid-string",
```

```
    "name": "production-client-001",
    "host": "192.168.1.100",
    "version": "1.0.0",
    "capabilities": ["docker_management", "system_monitoring"],
    "system_info": {
      "os": "linux",
      "arch": "amd64",
      "cpu_cores": 8,
      "memory_gb": 16,
      "disk_gb": 500
    },
    "tenant_code": "acme-corp"
  },
  "security": {
    "signature": "cryptographic-signature",
    "certificate_fingerprint": "sha256:abc123..."
  }
}
```

Task Execution Message

Copy

```
{
  "type": "task_execute",
  "id": "task-uuid-string",
  "timestamp": "2025-10-07T10:30:00Z",
  "payload": {
    "task_type": "docker_management",
    "action": "start_service",
    "parameters": {
      "service_name": "webapp",
      "image": "nginx:latest",
      "ports": ["80:8080"],
      "environment": {
        "ENV": "production"
      }
    },
    "timeout": 300,
    "priority": 1
  },
  "security": {
    "signature": "cryptographic-signature",
    "timestamp": "2025-10-07T10:30:00Z"
```

```
  }
}
```

Task Result Message

Copy

```
{
  "type": "task_result",
  "id": "task-uuid-string",
  "timestamp": "2025-10-07T10:32:30Z",
  "payload": {
    "success": true,
    "status": "completed",
    "output": "Service 'webapp' started successfully\nContainer ID: abc123d
    "exit_code": 0,
    "execution_time_ms": 2500,
    "resource_usage": {
      "cpu_percent": 15.5,
      "memory_mb": 128,
      "disk_io_mb": 50,
      "network_io_mb": 25
    }
  },
  "security": {
    "signature": "cryptographic-signature"
  }
}
```

Error Handling Protocol

```
{
  "type": "error",
  "id": "task-uuid-string",
  "timestamp": "2025-10-07T10:32:30Z",
  "payload": {
    "error_code": "DOCKER_SERVICE_FAILED",
    "error_message": "Failed to start service: image not found",
    "details": {
      "image": "nginx:invalid-tag",
      "registry_error": "manifest unknown"
    },
    "suggested_action": "Check image name and tag, ensure registry access",
```

```
    "recoverable": true
  },
  "security": {
    "signature": "cryptographic-signature"
  }
}
```

Security Implementation Details

Certificate-based Authentication

Copy

```
// SecurityManager handles all security operations
type SecurityManager struct {
    certificateStore  *CertificateStore
    jwtManager        *JWTManager
    rbacEngine        *RBACEngine
    auditLogger       *AuditLogger
    rateLimiter       *RateLimiter
}

// CertificateStore manages client certificates
type CertificateStore struct {
    caCertificate     *x509.Certificate
    caPrivateKey      *rsa.PrivateKey
    clientCerts       map[string]*x509.Certificate
    revocationList    map[string]time.Time
    mutex             sync.RWMutex
}

// JWTManager handles JWT token operations
type JWTManager struct {
    secretKey         []byte
    issuer            string
    defaultExpiry     time.Duration
    refreshExpiry     time.Duration
    blacklistedTokens map[string]time.Time
    mutex             sync.RWMutex
}

// RBACEngine implements role-based access control
type RBACEngine struct {
    roles             map[string]*Role
    permissions       map[string]*Permission
    userRoles         map[string][]string
    tenantPolicies    map[string]*TenantPolicy
    mutex             sync.RWMutex
}
```

Encryption Implementation

```
// MessageEncryption handles message-level encryption
type MessageEncryption struct {
    algorithm       string              // AES-256-GCM
    keyDerivation   string              // PBKDF2
    keyRotation     time.Duration       // 24 hours
    currentKey      []byte
    previousKeys    map[string][]byte
    mutex           sync.RWMutex
}

// EncryptMessage encrypts a message for transmission
func (me *MessageEncryption) EncryptMessage(data []byte, recipientID string
    // Generate random nonce
    nonce := make([]byte, 12)
    if _, err := rand.Read(nonce); err != nil {
        return nil, fmt.Errorf("failed to generate nonce: %w", err)
    }

    // Get current encryption key
    key := me.getCurrentKey()

    // Create AES-GCM cipher
    block, err := aes.NewCipher(key)
    if err != nil {
        return nil, fmt.Errorf("failed to create cipher: %w", err)
    }

    gcm, err := cipher.NewGCM(block)
    if err != nil {
        return nil, fmt.Errorf("failed to create GCM: %w", err)
    }

    // Encrypt the data
    ciphertext := gcm.Seal(nil, nonce, data, []byte(recipientID))

    return &EncryptedMessage{
        Ciphertext:   ciphertext,
        Nonce:        nonce,
        KeyID:        me.getCurrentKeyID(),
        Algorithm:    "AES-256-GCM",
        Timestamp:    time.Now(),
```

```
    }, nil
}
```

Audit Logging Implementation

```
// AuditLogger provides comprehensive audit logging
type AuditLogger struct {
    database        *mongo.Collection
    encryptionKey   []byte
    retentionDays   int
    asyncChannel    chan *AuditEvent
    batchSize       int
    batchTimeout    time.Duration
}

// AuditEvent represents an auditable event
type AuditEvent struct {
    ID              string                 `bson:"_id" json:"id"`
    Timestamp       time.Time              `bson:"timestamp" json:"timestamp
    EventType       string                 `bson:"event_type" json:"event_ty
    Action          string                 `bson:"action" json:"action"`
    Result          string                 `bson:"result" json:"result"`
    Actor           *AuditActor            `bson:"actor" json:"actor"`
    Target          *AuditTarget           `bson:"target" json:"target"`
    TenantCode      string                 `bson:"tenant_code" json:"tenant_
    IPAddress       string                 `bson:"ip_address" json:"ip_addre
    UserAgent       string                 `bson:"user_agent" json:"user_age
    SessionID       string                 `bson:"session_id" json:"session_
    Details         map[string]interface{} `bson:"details" json:"details"`
    Severity        string                 `bson:"severity" json:"severity"`
}

// LogEvent logs an audit event asynchronously
func (al *AuditLogger) LogEvent(event *AuditEvent) {
    // Add to async channel for batch processing
    select {
    case al.asyncChannel <- event:
        // Successfully queued
    default:
        // Channel full, log error and try sync
        log.Warn("Audit channel full, logging synchronously")
        al.logEventSync(event)
```

```
    }
}
```

Performance Optimization Techniques

Connection Pool Management

Copy

```
// ConnectionPool manages WebSocket connections efficiently
type ConnectionPool struct {
    maxConnections    int
    activeConnections map[string]*websocket.Conn
    connectionQueue   chan *websocket.Conn
    healthChecker     *HealthChecker
    loadBalancer      *LoadBalancer
    metrics           *ConnectionMetrics
    mutex             sync.RWMutex
}

// HealthChecker monitors connection health
type HealthChecker struct {
    interval          time.Duration
    timeout           time.Duration
    failureThreshold  int
    healthyThreshold  int
    checkQueue        chan string
}

// LoadBalancer distributes tasks across clients
type LoadBalancer struct {
    algorithm         string              // round_robin, least_connections,
    clientWeights     map[string]int
    connectionCounts map[string]int
    lastUsed          map[string]time.Time
    mutex             sync.RWMutex
}
```

Caching Strategy

```
// CacheManager handles multi-level caching
type CacheManager struct {
    l1Cache           *sync.Map           // In-memory cache
    l2Cache           *redis.Client       // Redis cache
    l3Cache           *mongo.Collection   // Database cache
```

```
    defaultTTL        time.Duration
    compressionEnabled bool
    metrics           *CacheMetrics
}

// CacheKey represents a cache key with metadata
type CacheKey struct {
    Key               string
    TenantCode        string
    Category          string
    TTL               time.Duration
    Compressed        bool
}

// Get retrieves a value from the cache hierarchy
func (cm *CacheManager) Get(key *CacheKey) (interface{}, error) {
    // Try L1 cache first (in-memory)
    if value, ok := cm.l1Cache.Load(key.Key); ok {
        cm.metrics.RecordHit("l1")
        return value, nil
    }

    // Try L2 cache (Redis)
    if value, err := cm.l2Cache.Get(context.Background(), key.Key).Result()
        cm.metrics.RecordHit("l2")
        // Store in L1 for next time
        cm.l1Cache.Store(key.Key, value)
        return value, nil
    }

    // Try L3 cache (Database)
    var cacheEntry CacheEntry
    filter := bson.M{"key": key.Key, "tenant_code": key.TenantCode}
    err := cm.l3Cache.FindOne(context.Background(), filter).Decode(&cacheEn
    if err == nil && cacheEntry.ExpiresAt.After(time.Now()) {
        cm.metrics.RecordHit("l3")
        // Store in L1 and L2 for next time
        cm.l1Cache.Store(key.Key, cacheEntry.Value)
        cm.l2Cache.Set(context.Background(), key.Key, cacheEntry.Value, key
        return cacheEntry.Value, nil
    }

    cm.metrics.RecordMiss()
    return nil, ErrCacheMiss
}
```

Message Batching and Compression

```
// MessageBatcher batches messages for efficient transmission
type MessageBatcher struct {
    batchSize        int
    batchTimeout     time.Duration
    compressionLevel int
    messageQueue     chan *Message
    batchQueue       chan []*Message
    compressor       *Compressor
}

// Compressor handles message compression
type Compressor struct {
    algorithm        string              // gzip, lz4, zstd
    level            int
    threshold        int                 // Minimum size to compress
    dictionary       []byte              // Compression dictionary
}

// BatchMessages groups messages for efficient transmission
func (mb *MessageBatcher) BatchMessages(messages []*Message) (*BatchedMessa
    batch := &BatchedMessage{
        ID:           generateBatchID(),
        Timestamp:    time.Now(),
        MessageCount: len(messages),
        Messages:     messages,
    }

    // Serialize batch
    data, err := json.Marshal(batch)
    if err != nil {
        return nil, fmt.Errorf("failed to serialize batch: %w", err)
    }

    // Compress if beneficial
    if len(data) > mb.compressor.threshold {
        compressed, err := mb.compressor.Compress(data)
        if err != nil {
            return nil, fmt.Errorf("failed to compress batch: %w", err)
        }
        batch.Data = compressed
        batch.Compressed = true
        batch.CompressionRatio = float64(len(data)) / float64(len(compresse
```

```
    } else {
        batch.Data = data
        batch.Compressed = false
    }

    return batch, nil
}
```

Testing and Quality Assurance

Unit Testing Framework

```
// TestSuite represents a comprehensive test suite
type TestSuite struct {
    name            string
    database        *TestDatabase
    server          *TestServer
    clients         []*TestClient
    cleanup         []func()
}

// TestDatabase provides isolated test database
type TestDatabase struct {
    client          *mongo.Client
    database        *mongo.Database
    collections     map[string]*mongo.Collection
    fixtures        map[string]interface{}
}

// TestServer provides test server instance
type TestServer struct {
    app             *App
    server          *httptest.Server
    wsServer        *httptest.Server
    config          *Config
}

// Integration test example
func TestTaskExecution(t *testing.T) {
    suite := NewTestSuite("task_execution")
    defer suite.Cleanup()

    // Setup test data
    client := suite.CreateTestClient("test-client-001")
    task := &Task{
        ID:          "test-task-001",
        Type:        "docker_management",
        ClientID:    client.ID,
        Priority:    1,
        Payload: TaskPayload{
            TaskType: "start_service",
```

```
            Parameters: map[string]interface{}{
                "service_name": "test-service",
                "image":        "nginx:alpine",
            },
        },
    }

    // Execute test
    result, err := suite.server.ExecuteTask(task)
    assert.NoError(t, err)
    assert.NotNil(t, result)
    assert.Equal(t, "running", result.Status)

    // Wait for completion
    finalResult, err := suite.WaitForTaskCompletion(task.ID, 30*time.Second
    assert.NoError(t, err)
    assert.Equal(t, "completed", finalResult.Status)
    assert.True(t, finalResult.Result.Success)
}
```

Load Testing Implementation

Copy

```
// LoadTester provides comprehensive load testing
type LoadTester struct {
    targetURL       string
    maxConnections  int
    testDuration    time.Duration
    messageRate     int
    metrics         *LoadTestMetrics
    clients         []*LoadTestClient
}

// LoadTestMetrics tracks performance metrics
type LoadTestMetrics struct {
    TotalRequests       int64
    SuccessfulRequests  int64
    FailedRequests      int64
    AverageLatency      time.Duration
    MaxLatency          time.Duration
    MinLatency          time.Duration
    ThroughputPerSecond float64
    ErrorRate           float64
    mutex               sync.RWMutex
```

```
}

// RunLoadTest executes a comprehensive load test
func (lt *LoadTester) RunLoadTest() (*LoadTestResult, error) {
    startTime := time.Now()

    // Create test clients
    for i := 0; i < lt.maxConnections; i++ {
        client := &LoadTestClient{
            ID:          fmt.Sprintf("load-test-client-%d", i),
            TargetURL:   lt.targetURL,
            MessageRate: lt.messageRate,
            Metrics:     lt.metrics,
        }
        lt.clients = append(lt.clients, client)

        // Start client in goroutine
        go client.Start()
    }

    // Run for specified duration
    time.Sleep(lt.testDuration)

    // Stop all clients
    for _, client := range lt.clients {
        client.Stop()
    }

    endTime := time.Now()

    return &LoadTestResult{
        Duration:          endTime.Sub(startTime),
        TotalConnections:  lt.maxConnections,
        Metrics:           lt.metrics.GetSnapshot(),
        ClientMetrics:     lt.getClientMetrics(),
    }, nil
}
```

RIS Client Implementation Details

Client Agent Architecture

```
// RISClient represents the main client application structure
type RISClient struct {
    Config              *ClientConfig              // Client configuration
    Connection          *websocket.Conn            // WebSocket connection
    DockerClient        *docker.Client             // Docker client for con
    TaskExecutor        *TaskExecutor              // Task execution engine
    HealthMonitor       *HealthMonitor             // Health monitoring sys
    SecurityManager     *ClientSecurityManager     // Security and authenti
    Logger              *logrus.Logger             // Structured logging
    MessageQueue        chan *Message              // Incoming message queu
    ResponseQueue       chan *Response             // Outgoing response que
    IsConnected         bool                       // Connection status
    LastHeartbeat       time.Time                  // Last heartbeat timest
    ClientID            string                     // Unique client identif
    TenantCode          string                     // Tenant identification
}

// ClientConfig holds all client configuration
type ClientConfig struct {
    ServerURL           string                     `json:"server_url"`
    ClientID            string                     `json:"client_id"`
    ClientName          string                     `json:"client_name"`
    TenantCode          string                     `json:"tenant_code"`
    TLS struct {
        CertFile        string                     `json:"cert_file"`
        KeyFile         string                     `json:"key_file"`
        CAFile          string                     `json:"ca_file"`
        InsecureSkipVerify bool                    `json:"insecure_skip_ver
    } `json:"tls"`
    Docker struct {
        Host            string                     `json:"host"`
        APIVersion      string                     `json:"api_version"`
        TLS             bool                       `json:"tls"`
    } `json:"docker"`
    Monitoring struct {
        HeartbeatInterval time.Duration            `json:"heartbeat_interv
        HealthCheckInterval time.Duration           `json:"health_check_int
        MetricsEnabled    bool                     `json:"metrics_enabled"`
```

```
    } `json:"monitoring"`
}
```

Task Execution Framework

Copy

```
// TaskExecutor handles all task execution
type TaskExecutor struct {
    dockerClient        *docker.Client
    httpClient          *http.Client
    systemExecutor      *SystemExecutor
    maxConcurrentTasks int
    runningTasks        map[string]*RunningTask
    taskQueue           chan *TaskRequest
    logger              *logrus.Logger
    mutex               sync.RWMutex
}

// RunningTask represents a task being executed
type RunningTask struct {
    ID                string                 `json:"id"`
    Type              string                 `json:"type"`
    Status            string                 `json:"status"`
    StartTime         time.Time              `json:"start_time"`
    Context           context.Context        `json:"-"`
    CancelFunc        context.CancelFunc     `json:"-"`
    Progress          int                    `json:"progress"`
    Output            []string               `json:"output"`
    Error             string                 `json:"error,omitempty"`
}

// TaskRequest represents an incoming task request
type TaskRequest struct {
    ID                string                 `json:"id"`
    Type              string                 `json:"type"`
    Parameters        map[string]interface{} `json:"parameters"`
    Timeout           time.Duration          `json:"timeout"`
    Priority          int                    `json:"priority"`
    ResponseChannel chan *TaskResponse      `json:"-"`
}

// ExecuteTask processes a task request
func (te *TaskExecutor) ExecuteTask(req *TaskRequest) *TaskResponse {
    te.mutex.Lock()
```

```
    defer te.mutex.Unlock()

    // Create context with timeout
    ctx, cancel := context.WithTimeout(context.Background(), req.Timeout)

    task := &RunningTask{
        ID:         req.ID,
        Type:       req.Type,
        Status:     "running",
        StartTime:  time.Now(),
        Context:    ctx,
        CancelFunc: cancel,
        Progress:   0,
    }

    te.runningTasks[req.ID] = task

    // Execute based on task type
    var response *TaskResponse
    switch req.Type {
    case "docker_management":
        response = te.executeDockerTask(task, req.Parameters)
    case "http_request":
        response = te.executeHTTPTask(task, req.Parameters)
    case "system_command":
        response = te.executeSystemTask(task, req.Parameters)
    default:
        response = &TaskResponse{
            TaskID:  req.ID,
            Success: false,
            Error:   fmt.Sprintf("Unknown task type: %s", req.Type),
        }
    }

    // Cleanup
    delete(te.runningTasks, req.ID)
    cancel()

    return response
}
```

Docker Management Implementation

```
// DockerManager handles Docker operations
type DockerManager struct {
    client         *docker.Client
    logger         *logrus.Logger
    timeout        time.Duration
}

// executeDockerTask handles Docker-related tasks
func (te *TaskExecutor) executeDockerTask(task *RunningTask, params map[str
    action, ok := params["action"].(string)
    if !ok {
        return &TaskResponse{
            TaskID:  task.ID,
            Success: false,
            Error:   "Missing 'action' parameter",
        }
    }

    switch action {
    case "start_service":
        return te.startDockerService(task, params)
    case "stop_service":
        return te.stopDockerService(task, params)
    case "restart_service":
        return te.restartDockerService(task, params)
    case "get_status":
        return te.getDockerServiceStatus(task, params)
    case "pull_image":
        return te.pullDockerImage(task, params)
    case "deploy_stack":
        return te.deployDockerStack(task, params)
    default:
        return &TaskResponse{
            TaskID:  task.ID,
            Success: false,
            Error:   fmt.Sprintf("Unknown Docker action: %s", action),
        }
    }
}

// startDockerService starts a Docker service
func (te *TaskExecutor) startDockerService(task *RunningTask, params map[st
    serviceName, _ := params["service_name"].(string)
    image, _ := params["image"].(string)
```

```
    if serviceName == "" || image == "" {
        return &TaskResponse{
            TaskID:  task.ID,
            Success: false,
            Error:   "Missing required parameters: service_name, image",
        }
    }

    // Parse additional parameters
    ports, _ := params["ports"].([]interface{})
    environment, _ := params["environment"].(map[string]interface{})
    volumes, _ := params["volumes"].([]interface{})

    // Create container configuration
    config := &container.Config{
        Image: image,
        Env:   te.buildEnvironmentList(environment),
    }

    hostConfig := &container.HostConfig{
        PortBindings: te.buildPortBindings(ports),
        Binds:        te.buildVolumeBindings(volumes),
        RestartPolicy: container.RestartPolicy{
            Name: "unless-stopped",
        },
    }

    // Create and start container
    resp, err := te.dockerClient.ContainerCreate(
        task.Context,
        config,
        hostConfig,
        nil,
        nil,
        serviceName,
    )

    if err != nil {
        return &TaskResponse{
            TaskID:  task.ID,
            Success: false,
            Error:   fmt.Sprintf("Failed to create container: %v", err),
        }
    }

    err = te.dockerClient.ContainerStart(task.Context, resp.ID, types.Conta
```

```
    if err != nil {
        return &TaskResponse{
            TaskID:  task.ID,
            Success: false,
            Error:   fmt.Sprintf("Failed to start container: %v", err),
        }
    }

    task.Progress = 100
    task.Status = "completed"

    return &TaskResponse{
        TaskID:        task.ID,
        Success:       true,
        Output:        fmt.Sprintf("Service '%s' started successfully. Conta
        ContainerID:   resp.ID,
        ExecutionTime: time.Since(task.StartTime),
    }
}
```

HTTP Request Implementation

```
// executeHTTPTask handles HTTP requests
func (te *TaskExecutor) executeHTTPTask(task *RunningTask, params map[strin
    url, ok := params["url"].(string)
    if !ok {
        return &TaskResponse{
            TaskID:  task.ID,
            Success: false,
            Error:   "Missing 'url' parameter",
        }
    }

    method, _ := params["method"].(string)
    if method == "" {
        method = "GET"
    }

    // Parse headers
    headers, _ := params["headers"].(map[string]interface{})

    // Parse body
    var body io.Reader
```

```
    if bodyData, exists := params["body"]; exists {
        if bodyStr, ok := bodyData.(string); ok {
            body = strings.NewReader(bodyStr)
        }
    }

    // Create HTTP request
    req, err := http.NewRequestWithContext(task.Context, method, url, body)
    if err != nil {
        return &TaskResponse{
            TaskID:  task.ID,
            Success: false,
            Error:   fmt.Sprintf("Failed to create HTTP request: %v", err),
        }
    }

    // Set headers
    for key, value := range headers {
        if valueStr, ok := value.(string); ok {
            req.Header.Set(key, valueStr)
        }
    }

    // Execute request
    client := &http.Client{
        Timeout: 30 * time.Second,
    }

    resp, err := client.Do(req)
    if err != nil {
        return &TaskResponse{
            TaskID:  task.ID,
            Success: false,
            Error:   fmt.Sprintf("HTTP request failed: %v", err),
        }
    }
    defer resp.Body.Close()

    // Read response body
    responseBody, err := io.ReadAll(resp.Body)
    if err != nil {
        return &TaskResponse{
            TaskID:  task.ID,
            Success: false,
            Error:   fmt.Sprintf("Failed to read response: %v", err),
        }
```

```
    }

    task.Progress = 100
    task.Status = "completed"

    success := resp.StatusCode >= 200 && resp.StatusCode < 300

    return &TaskResponse{
        TaskID:        task.ID,
        Success:       success,
        Output:        string(responseBody),
        HTTPStatus:    resp.StatusCode,
        HTTPHeaders:   resp.Header,
        ExecutionTime: time.Since(task.StartTime),
    }
}
```

Health Monitoring System

Copy

```
// HealthMonitor provides comprehensive health monitoring
type HealthMonitor struct {
    client          *RISClient
    interval        time.Duration
    metrics         *HealthMetrics
    dockerClient    *docker.Client
    logger          *logrus.Logger
    stopChannel     chan bool
}

// HealthMetrics contains all health-related metrics
type HealthMetrics struct {
    SystemLoad      float64           `json:"system_load"`
    MemoryUsage     MemoryMetrics     `json:"memory_usage"`
    DiskUsage       DiskMetrics       `json:"disk_usage"`
    NetworkUsage    NetworkMetrics    `json:"network_usage"`
    DockerStats     DockerMetrics     `json:"docker_stats"`
    TaskMetrics     TaskMetrics       `json:"task_metrics"`
    LastUpdated     time.Time         `json:"last_updated"`
}

// MemoryMetrics contains memory usage information
type MemoryMetrics struct {
    Total           uint64            `json:"total_bytes"`
```

```
    Used                uint64                 `json:"used_bytes"`
    Available           uint64                 `json:"available_bytes"`
    UsagePercent        float64                `json:"usage_percent"`
}

// collectHealthMetrics gathers all health metrics
func (hm *HealthMonitor) collectHealthMetrics() *HealthMetrics {
    metrics := &HealthMetrics{
        LastUpdated: time.Now(),
    }

    // Collect system load
    if load, err := hm.getSystemLoad(); err == nil {
        metrics.SystemLoad = load
    }

    // Collect memory metrics
    if memInfo, err := hm.getMemoryInfo(); err == nil {
        metrics.MemoryUsage = *memInfo
    }

    // Collect disk metrics
    if diskInfo, err := hm.getDiskInfo(); err == nil {
        metrics.DiskUsage = *diskInfo
    }

    // Collect Docker metrics
    if dockerInfo, err := hm.getDockerMetrics(); err == nil {
        metrics.DockerStats = *dockerInfo
    }

    // Collect task metrics
    metrics.TaskMetrics = hm.getTaskMetrics()

    return metrics
}

// getSystemLoad retrieves current system load
func (hm *HealthMonitor) getSystemLoad() (float64, error) {
    loadavg, err := load.Avg()
    if err != nil {
        return 0, err
    }
    return loadavg.Load1, nil
}
```

```
// getMemoryInfo retrieves memory usage information
func (hm *HealthMonitor) getMemoryInfo() (*MemoryMetrics, error) {
    vmStat, err := mem.VirtualMemory()
    if err != nil {
        return nil, err
    }

    return &MemoryMetrics{
        Total:       vmStat.Total,
        Used:        vmStat.Used,
        Available:   vmStat.Available,
        UsagePercent: vmStat.UsedPercent,
    }, nil
}

// sendHealthUpdate sends health metrics to server
func (hm *HealthMonitor) sendHealthUpdate() error {
    metrics := hm.collectHealthMetrics()

    message := &Message{
        Type:      "health_update",
        ID:        generateMessageID(),
        Timestamp: time.Now(),
        Payload:   metrics,
    }

    return hm.client.SendMessage(message)
}
```

Client Configuration Management

```
// ConfigurationManager handles dynamic configuration updates
type ConfigurationManager struct {
    configPath    string
    currentConfig *ClientConfig
    callbacks     map[string][]ConfigCallback
    logger        *logrus.Logger
    mutex         sync.RWMutex
}

// ConfigCallback represents a configuration change callback
type ConfigCallback func(key string, oldValue, newValue interface{}) error
```

```
// LoadConfiguration loads configuration from file
func (cm *ConfigurationManager) LoadConfiguration() error {
    cm.mutex.Lock()
    defer cm.mutex.Unlock()

    data, err := os.ReadFile(cm.configPath)
    if err != nil {
        return fmt.Errorf("failed to read config file: %w", err)
    }

    var config ClientConfig
    if err := json.Unmarshal(data, &config); err != nil {
        return fmt.Errorf("failed to parse config: %w", err)
    }

    // Validate configuration
    if err := cm.validateConfiguration(&config); err != nil {
        return fmt.Errorf("invalid configuration: %w", err)
    }

    cm.currentConfig = &config
    return nil
}

// UpdateConfiguration updates configuration dynamically
func (cm *ConfigurationManager) UpdateConfiguration(updates map[string]inte
    cm.mutex.Lock()
    defer cm.mutex.Unlock()

    oldConfig := *cm.currentConfig

    // Apply updates
    for key, value := range updates {
        if err := cm.applyConfigUpdate(key, value); err != nil {
            return fmt.Errorf("failed to apply update %s: %w", key, err)
        }
    }

    // Validate updated configuration
    if err := cm.validateConfiguration(cm.currentConfig); err != nil {
        // Rollback on validation failure
        cm.currentConfig = &oldConfig
        return fmt.Errorf("configuration validation failed: %w", err)
    }

    // Save to file
```

```
    if err := cm.saveConfiguration(); err != nil {
        cm.currentConfig = &oldConfig
        return fmt.Errorf("failed to save configuration: %w", err)
    }

    // Notify callbacks
    for key, value := range updates {
        cm.notifyCallbacks(key, nil, value)
    }

    return nil
}

// applyConfigUpdate applies a single configuration update
func (cm *ConfigurationManager) applyConfigUpdate(key string, value interfa
    switch key {
    case "server_url":
        if url, ok := value.(string); ok {
            cm.currentConfig.ServerURL = url
        } else {
            return fmt.Errorf("invalid type for server_url")
        }
    case "heartbeat_interval":
        if duration, ok := value.(string); ok {
            if d, err := time.ParseDuration(duration); err == nil {
                cm.currentConfig.Monitoring.HeartbeatInterval = d
            } else {
                return fmt.Errorf("invalid duration format: %s", duration)
            }
        } else {
            return fmt.Errorf("invalid type for heartbeat_interval")
        }
    case "health_check_interval":
        if duration, ok := value.(string); ok {
            if d, err := time.ParseDuration(duration); err == nil {
                cm.currentConfig.Monitoring.HealthCheckInterval = d
            } else {
                return fmt.Errorf("invalid duration format: %s", duration)
            }
        } else {
            return fmt.Errorf("invalid type for health_check_interval")
        }
    default:
        return fmt.Errorf("unknown configuration key: %s", key)
    }
```

```
    return nil
}
```

Testing and Quality Assurance

```
// ClientTestSuite provides comprehensive client testing
type ClientTestSuite struct {
    client          *RISClient
    mockServer      *MockRISServer
    testDocker      *TestDockerClient
    tempDir         string
    cleanup         []func()
}

// MockRISServer provides mock server for testing
type MockRISServer struct {
    server          *httptest.Server
    wsServer        *websocket.Upgrader
    connections     []*websocket.Conn
    receivedMessages []*Message
    responses       map[string]*Message
    mutex           sync.RWMutex
}

// TestDockerClient provides mock Docker client
type TestDockerClient struct {
    containers      map[string]*MockContainer
    images          map[string]*MockImage
    networks        map[string]*MockNetwork
    volumes         map[string]*MockVolume
}

// Integration test example
func TestClientTaskExecution(t *testing.T) {
    suite := NewClientTestSuite()
    defer suite.Cleanup()

    // Setup test scenario
    suite.mockServer.SetResponse("task_execute", &Message{
        Type: "task_result",
        Payload: map[string]interface{}{
            "success": true,
            "output":  "Task completed successfully",
```

```
        },
    })

    // Execute test
    taskReq := &TaskRequest{
        ID:   "test-task-001",
        Type: "docker_management",
        Parameters: map[string]interface{}{
            "action":       "start_service",
            "service_name": "test-nginx",
            "image":        "nginx:alpine",
        },
        Timeout: 30 * time.Second,
    }

    response := suite.client.TaskExecutor.ExecuteTask(taskReq)

    // Verify results
    assert.True(t, response.Success)
    assert.Contains(t, response.Output, "started successfully")
    assert.NotEmpty(t, response.ContainerID)
}

// Performance test example
func TestClientPerformance(t *testing.T) {
    suite := NewClientTestSuite()
    defer suite.Cleanup()

    // Setup performance test
    const numTasks = 100
    const concurrency = 10

    var wg sync.WaitGroup
    results := make(chan *TaskResponse, numTasks)

    startTime := time.Now()

    // Execute concurrent tasks
    for i := 0; i < concurrency; i++ {
        wg.Add(1)
        go func(workerID int) {
            defer wg.Done()

            for j := 0; j < numTasks/concurrency; j++ {
                taskReq := &TaskRequest{
                    ID:   fmt.Sprintf("perf-task-%d-%d", workerID, j),
```

```
                Type: "http_request",
                Parameters: map[string]interface{}{
                    "url":    "http://httpbin.org/get",
                    "method": "GET",
                },
                Timeout: 10 * time.Second,
            }

            response := suite.client.TaskExecutor.ExecuteTask(taskReq)
            results <- response
        }
    }(i)
}

wg.Wait()
close(results)

duration := time.Since(startTime)

// Analyze results
successCount := 0
var totalExecutionTime time.Duration

for response := range results {
    if response.Success {
        successCount++
    }
    totalExecutionTime += response.ExecutionTime
}

successRate := float64(successCount) / float64(numTasks) * 100
avgExecutionTime := totalExecutionTime / time.Duration(numTasks)
throughput := float64(numTasks) / duration.Seconds()

t.Logf("Performance Results:")
t.Logf("  Total Duration: %v", duration)
t.Logf("  Success Rate: %.2f%%", successRate)
t.Logf("  Average Execution Time: %v", avgExecutionTime)
t.Logf("  Throughput: %.2f tasks/second", throughput)

// Assert performance requirements
assert.Greater(t, successRate, 95.0, "Success rate should be > 95%")
assert.Less(t, avgExecutionTime, 5*time.Second, "Average execution time
assert.Greater(t, throughput, 10.0, "Throughput should be > 10 tasks/se
}
```

Version Information

COMPONENT	VERSION	GO VERSION	DEPENDENCIES
RIS Client Core	1.0.0	1.21+	Gorilla WebSocket, Docker Client
Task Executor	1.0.0	1.21+	Docker API, HTTP Client
Health Monitor	1.0.0	1.21+	gopsutil, system metrics
Security Module	1.0.0	1.21+	Crypto/TLS, Certificate handling
Configuration Manager	1.0.0	1.21+	JSON, File I/O