Usage Examples

This document provides practical examples of using the UBOM Workflow Engine’s pluggable work execution system.

Basic Usage

Setting Up the Enhanced Registry

```
package main

import (
    "context"
    "log"

    "github.com/ubom/workflow/layer0"
    "github.com/ubom/workflow/executors"
    "github.com/ubom/workflow/executors/docker"
    "github.com/ubom/workflow/executors/grpc"
    "github.com/ubom/workflow/executors/serverless"
    "github.com/ubom/workflow/schemas"
)

func main() {
    // Create enhanced registry
    registry := executors.NewEnhancedWorkRegistry()

    // Create schema validator
    validator := schemas.NewSchemaValidator()

    // Register built-in executors
    setupBuiltinExecutors(registry, validator)

    // Execute some work
    executeExampleWork(registry, validator)
}

func setupBuiltinExecutors(registry *executors.EnhancedWorkRegistry, validator *schem-
as.SchemaValidator) {
    // Docker executor
    dockerClient := NewDockerClient() // Your Docker client implementation
    logger := NewLogger()             // Your logger implementation
    dockerExecutor := docker.NewDockerExecutor(dockerClient, logger)

    registry.RegisterExecutor(docker.WorkTypeDocker, dockerExecutor)
    validator.RegisterSchema(docker.WorkTypeDocker, dockerExecutor.GetSchema())

    // gRPC executor
    connectionPool := NewGRPCConnectionPool() // Your connection pool implementation
    grpcExecutor := grpc.NewGRPCExecutor(connectionPool, logger)

    registry.RegisterExecutor(grpc.WorkTypeGRPC, grpcExecutor)
    validator.RegisterSchema(grpc.WorkTypeGRPC, grpcExecutor.GetSchema())

    // Serverless executor
    providers := map[string]serverless.ServerlessProvider{
        "aws": NewAWSProvider(),   // Your AWS provider implementation
        "gcp": NewGCPProvider(),   // Your GCP provider implementation
    }
    serverlessExecutor := serverless.NewServerlessExecutor(providers, logger)

    registry.RegisterExecutor(serverless.WorkTypeServerless, serverlessExecutor)
    validator.RegisterSchema(serverless.WorkTypeServerless, serverlessExecut-
or.GetSchema())
}
```

Docker Executor Examples

Simple Container Execution

```
func executeDockerWork() {
    // Create work
    work := layer0.NewWork("docker-example", docker.WorkTypeDocker, "Python Script Execution")

    // Configure Docker execution
    config := work.GetConfiguration()
    config.Parameters["executor_config"] = map[string]interface{}{
        "image": "python:3.9-slim",
        "command": []string{"python", "-c"},
        "args": []string{"print('Hello from Docker!')"},
        "environment": map[string]string{
            "PYTHONPATH": "/app",
        },
    }

    // Update work with configuration
    work = layer0.Work{
        ID:            work.GetID(),
        Type:          work.GetType(),
        Status:        work.GetStatus(),
        Priority:      work.GetPriority(),
        Metadata:      work.GetMetadata(),
        Configuration: config,
        Input:         work.GetInput(),
        Output:        work.GetOutput(),
        Error:         work.GetError(),
    }

    // Execute
    executor, _ := registry.GetExecutor(docker.WorkTypeDocker)
    result, err := executor.Execute(context.Background(), work, &layer0.Context{})

    if err != nil {
        log.Printf("Execution failed: %v", err)
        return
    }

    log.Printf("Execution result: %+v", result)
}
```

Data Processing with Volumes

```
func executeDataProcessing() {
    work := layer0.NewWork("data-processing", docker.WorkTypeDocker, "Data Processing
Job")

    config := work.GetConfiguration()
    config.Parameters["executor_config"] = map[string]interface{}{
        "image": "my-data-processor:latest",
        "command": []string{"python", "process_data.py"},
        "volumes": []map[string]interface{}{
            {
                "source": "/host/data/input",
                "target": "/app/input",
                "read_only": true,
            },
            {
                "source": "/host/data/output",
                "target": "/app/output",
                "read_only": false,
            },
        },
        "environment": map[string]string{
            "INPUT_FORMAT":  "csv",
            "OUTPUT_FORMAT": "json",
            "BATCH_SIZE":    "1000",
        },
        "resources": map[string]interface{}{
            "cpu_limit":    "2.0",
            "memory_limit": "4g",
        },
    }

    work = updateWorkConfig(work, config)

    // Execute with input data
    work = work.SetInput(map[string]interface{}{
        "dataset": "customer_data_2023.csv",
        "filters": []string{"active=true", "region=us-east"},
    })

    executor, _ := registry.GetExecutor(docker.WorkTypeDocker)
    result, err := executor.Execute(context.Background(), work, &layer0.Context{})

    handleResult(result, err)
}
```

gRPC Executor Examples

Simple Service Call

```
func executeGRPCWork() {
    work := layer0.NewWork("grpc-example", grpc.WorkTypeGRPC, "User Service Call")

    config := work.GetConfiguration()
    config.Parameters["executor_config"] = map[string]interface{}{
        "endpoint": "user-service:9090",
        "method":   "userservice.UserService/GetUser",
        "timeout":  "30s",
        "metadata": map[string]string{
            "authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
            "request-id":    "req-12345",
        },
    }

    work = updateWorkConfig(work, config)
    work = work.SetInput(map[string]interface{}{
        "user_id": "12345",
    })

    executor, _ := registry.GetExecutor(grpc.WorkTypeGRPC)
    result, err := executor.Execute(context.Background(), work, &layer0.Context{})

    handleResult(result, err)
}
```

Secure gRPC with Retry

```
func executeSecureGRPCWork() {
    work := layer0.NewWork("secure-grpc", grpc.WorkTypeGRPC, "Secure Payment Pro-
cessing")

    config := work.GetConfiguration()
    config.Parameters["executor_config"] = map[string]interface{}{
        "endpoint": "payment-service.example.com:443",
        "method":   "payment.PaymentService/ProcessPayment",
        "timeout":  "60s",
        "tls": map[string]interface{}{
            "enabled":     true,
            "server_name": "payment-service.example.com",
            "ca_file":     "/etc/ssl/certs/ca.pem",
        },
        "retry": map[string]interface{}{
            "max_attempts":  5,
            "initial_delay": "2s",
            "max_delay":     "30s",
            "multiplier":    2.0,
            "jitter":        true,
        },
        "metadata": map[string]string{
            "api-key":    "your-api-key",
            "idempotency-key": "payment-12345",
        },
    }

    work = updateWorkConfig(work, config)
    work = work.SetInput(map[string]interface{}{
        "amount":   10000, // $100.00 in cents
        "currency": "USD",
        "payment_method": map[string]interface{}{
            "type": "credit_card",
            "card_token": "tok_visa_4242",
        },
        "customer_id": "cust_12345",
    })

    executor, _ := registry.GetExecutor(grpc.WorkTypeGRPC)
    result, err := executor.Execute(context.Background(), work, &layer0.Context{})

    handleResult(result, err)
}
```

Serverless Executor Examples

AWS Lambda Function

```
func executeAWSLambda() {
    work := layer0.NewWork("aws-lambda", serverless.WorkTypeServerless, "Image Pro-
cessing Lambda")

    config := work.GetConfiguration()
    config.Parameters["executor_config"] = map[string]interface{}{
        "provider": "aws",
        "function": "image-processor",
        "region":   "us-east-1",
        "credentials": map[string]interface{}{
            "type":       "aws_iam",
            "access_key": "AKIAIOSFODNN7EXAMPLE",
            "secret_key": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
        },
        "timeout": "300s", // 5 minutes
        "memory":  512,    // 512 MB
        "environment": map[string]string{
            "S3_BUCKET": "my-images-bucket",
            "LOG_LEVEL": "INFO",
        },
    }

    work = updateWorkConfig(work, config)
    work = work.SetInput(map[string]interface{}{
        "image_url": "https://i.ytimg.com/vi/3aUq_jHyYfY/maxresdefault.jpg",
        "operations": []string{"resize", "watermark", "compress"},
        "output_format": "webp",
    })

    executor, _ := registry.GetExecutor(serverless.WorkTypeServerless)
    result, err := executor.Execute(context.Background(), work, &layer0.Context{})

    handleResult(result, err)
}
```

Google Cloud Function

```
func executeGCPFunction() {
    work := layer0.NewWork("gcp-function", serverless.WorkTypeServerless, "Text Analysis Function")

    config := work.GetConfiguration()
    config.Parameters["executor_config"] = map[string]interface{}{
        "provider": "gcp",
        "function": "text-analyzer",
        "region":   "us-central1",
        "credentials": map[string]interface{}{
            "type":       "service_account",
            "project_id": "my-project-123456",
            "key_file":   "/path/to/service-account.json",
        },
        "timeout": "120s",
        "memory":  256,
    }

    work = updateWorkConfig(work, config)
    work = work.SetInput(map[string]interface{}{
        "text": "This is a sample text for sentiment analysis and entity extraction.",
        "language": "en",
        "features": []string{"sentiment", "entities", "syntax"},
    })

    executor, _ := registry.GetExecutor(serverless.WorkTypeServerless)
    result, err := executor.Execute(context.Background(), work, &layer0.Context{})

    handleResult(result, err)
}
```

Azure Function

```
func executeAzureFunction() {
    work := layer0.NewWork("azure-function", serverless.WorkTypeServerless, "Document
Converter")

    config := work.GetConfiguration()
    config.Parameters["executor_config"] = map[string]interface{}{
        "provider": "azure",
        "function": "document-converter",
        "credentials": map[string]interface{}{
            "type": "managed_identity",
        },
        "timeout": "180s",
        "memory":  128,
    }

    work = updateWorkConfig(work, config)
    work = work.SetInput(map[string]interface{}{
        "document_url": "https://storage.example.com/docs/report.docx",
        "output_format": "pdf",
        "options": map[string]interface{}{
            "include_metadata": true,
            "compress": true,
        },
    })

    executor, _ := registry.GetExecutor(serverless.WorkTypeServerless)
    result, err := executor.Execute(context.Background(), work, &layer0.Context{})

    handleResult(result, err)
}
```

Plugin System Examples

Loading and Using Plugins

```go
func setupPluginSystem() {
    // Create plugin loader
    logger := NewLogger()
    loader := plugins.NewDefaultPluginLoader(logger)

    // Load plugins from directory
    err := loader.LoadPluginsFromDirectory("/opt/workflow/plugins")
    if err != nil {
        log.Printf("Failed to load plugins: %v", err)
    }

    // Register a custom plugin
    factory := func() plugins.ExternalWorkPlugin {
        return NewCustomPlugin()
    }

    err = loader.RegisterPlugin("custom-plugin", factory)
    if err != nil {
        log.Printf("Failed to register plugin: %v", err)
    }

    // Initialize plugins
    pluginConfig := map[string]interface{}{
        "api_endpoint": "https://api.example.com",
        "timeout":      "30s",
        "max_retries":  3,
    }

    err = loader.InitializePlugin("custom-plugin", pluginConfig)
    if err != nil {
        log.Printf("Failed to initialize plugin: %v", err)
    }

    // Use plugin
    plugin, err := loader.GetPlugin("custom-plugin")
    if err != nil {
        log.Printf("Failed to get plugin: %v", err)
        return
    }

    // Execute work with plugin
    work := createCustomWork()
    result, err := plugin.Execute(context.Background(), work, &layer0.Context{})
    handleResult(result, err)

    // Health check
    err = plugin.HealthCheck()
    if err != nil {
        log.Printf("Plugin health check failed: %v", err)
    }

    // Shutdown when done
    defer func() {
        err := loader.ShutdownPlugin("custom-plugin")
        if err != nil {
            log.Printf("Failed to shutdown plugin: %v", err)
        }
    }()
}
```

Custom Plugin Example

```go
// Example of a simple HTTP API plugin
type HTTPAPIPlugin struct {
    *plugins.BasePlugin
    client *http.Client
}

func NewHTTPAPIPlugin() plugins.ExternalWorkPlugin {
    info := plugins.PluginInfo{
        Name:        "http-api-plugin",
        Version:     "1.0.0",
        Author:      "Example Corp",
        Description: "Plugin for making HTTP API calls",
        WorkTypes:   []string{"http_api"},
        APIVersion:  "1.0",
    }

    metadata := executors.WorkMetadata{
        Name:        "HTTP API Plugin",
        Version:     "1.0.0",
        Author:      "Example Corp",
        Description: "Makes HTTP API calls",
        WorkTypes:   []string{"http_api"},
    }

    schema := executors.WorkSchema{
        JSONSchema: `{
            "type": "object",
            "required": ["url", "method"],
            "properties": {
                "url": {"type": "string"},
                "method": {"type": "string", "enum": ["GET", "POST", "PUT", "DELETE"]},
                "headers": {"type": "object"},
                "timeout": {"type": "string", "pattern": "^[0-9]+[smh]$"}
            }
        }`,
        Examples: []executors.WorkDefinition{
            {
                Name: "Simple GET Request",
                Configuration: map[string]interface{}{
                    "executor_config": map[string]interface{}{
                        "url":    "https://api.example.com/users",
                        "method": "GET",
                        "headers": map[string]string{
                            "Authorization": "Bearer token123",
                        },
                    },
                },
            },
        },
        Documentation: "HTTP API plugin for making REST API calls",
    }

    return &HTTPAPIPlugin{
        BasePlugin: plugins.NewBasePlugin(info, metadata, schema),
    }
}

func (p *HTTPAPIPlugin) Execute(ctx context.Context, work layer0.Work, workContext *lay
er0.Context) (executors.WorkResult, error) {
    startTime := time.Now()

    // Parse configuration
```

```
    config, err := p.parseConfig(work)
    if err != nil {
        return executors.WorkResult{
            Success: false,
            Error:   err.Error(),
            Metrics: executors.ExecutionMetrics{
                StartTime: startTime,
                EndTime:   time.Now(),
                Duration:  time.Since(startTime),
            },
        }, err
    }

    // Make HTTP request
    response, err := p.makeRequest(ctx, config, work.GetInput())
    endTime := time.Now()

    result := executors.WorkResult{
        Metrics: executors.ExecutionMetrics{
            StartTime: startTime,
            EndTime:   endTime,
            Duration:  endTime.Sub(startTime),
        },
    }

    if err != nil {
        result.Success = false
        result.Error = err.Error()
        return result, err
    }

    result.Success = true
    result.Outputs = map[string]interface{}{
        "response": response,
    }

    return result, nil
}

func (p *HTTPAPIPlugin) Initialize(config map[string]interface{}) error {
    timeout := 30 * time.Second
    if t, ok := config["timeout"].(string); ok {
        if parsed, err := time.ParseDuration(t); err == nil {
            timeout = parsed
        }
    }

    p.client = &http.Client{
        Timeout: timeout,
    }

    return nil
}

func (p *HTTPAPIPlugin) Shutdown() error {
    if p.client != nil {
        p.client.CloseIdleConnections()
    }
    return nil
}

func (p *HTTPAPIPlugin) HealthCheck() error {
    if p.client == nil {
```

```
        return fmt.Errorf("HTTP client not initialized")
    }
    return nil
}

func (p *HTTPAPIPlugin) CanExecute(workType layer0.WorkType) bool {
    return workType == "http_api"
}

func (p *HTTPAPIPlugin) GetSupportedTypes() []layer0.WorkType {
    return []layer0.WorkType{"http_api"}
}

func (p *HTTPAPIPlugin) Validate(work layer0.Work) error {
    if work.GetType() != "http_api" {
        return fmt.Errorf("unsupported work type: %s", work.GetType())
    }

    _, err := p.parseConfig(work)
    return err
}
```

Workflow Integration

Complete Workflow Example

```
func executeCompleteWorkflow() {
    // Setup
    registry := executors.NewEnhancedWorkRegistry()
    validator := schemas.NewSchemaValidator()
    setupBuiltinExecutors(registry, validator)

    // Create a multi-step workflow
    steps := []layer0.Work{
        createDataExtractionWork(),    // Docker: Extract data from source
        createDataValidationWork(),    // gRPC: Validate data quality
        createDataProcessingWork(),    // Serverless: Process and transform
        createNotificationWork(),      // Custom plugin: Send notifications
    }

    // Execute workflow steps
    var results []executors.WorkResult

    for i, work := range steps {
        log.Printf("Executing step %d: %s", i+1, work.GetMetadata().Name)

        // Validate work
        if err := validator.ValidateWork(work); err != nil {
            log.Printf("Validation failed for step %d: %v", i+1, err)
            continue
        }

        // Get executor
        executor, err := registry.GetExecutor(work.GetType())
        if err != nil {
            log.Printf("No executor found for step %d: %v", i+1, err)
            continue
        }

        // Execute
        result, err := executor.Execute(context.Background(), work, &layer0.Context{})
        if err != nil {
            log.Printf("Execution failed for step %d: %v", i+1, err)
            continue
        }

        results = append(results, result)

        // Use output from previous step as input for next step
        if i < len(steps)-1 && result.Success {
            steps[i+1] = steps[i+1].SetInput(result.Outputs)
        }

        log.Printf("Step %d completed successfully", i+1)
    }

    // Print workflow summary
    printWorkflowSummary(results)
}
```

Utility Functions

```
func updateWorkConfig(work layer0.Work, config layer0.WorkConfiguration) layer0.Work {
    return layer0.Work{
        ID:            work.GetID(),
        Type:          work.GetType(),
        Status:        work.GetStatus(),
        Priority:      work.GetPriority(),
        Metadata:      work.GetMetadata(),
        Configuration: config,
        Input:         work.GetInput(),
        Output:        work.GetOutput(),
        Error:         work.GetError(),
    }
}

func handleResult(result executors.WorkResult, err error) {
    if err != nil {
        log.Printf("Execution failed: %v", err)
        return
    }

    if result.Success {
        log.Printf("Execution successful:")
        log.Printf("  Duration: %v", result.Metrics.Duration)
        log.Printf("  Outputs: %+v", result.Outputs)

        if len(result.Logs) > 0 {
            log.Printf("  Logs:")
            for _, logEntry := range result.Logs {
                log.Printf("    [%s] %s: %s", logEntry.Level, logEntry.Source, lo-
gEntry.Message)
            }
        }
    } else {
        log.Printf("Execution failed: %s", result.Error)
    }
}

func printWorkflowSummary(results []executors.WorkResult) {
    successful := 0
    totalDuration := time.Duration(0)

    for _, result := range results {
        if result.Success {
            successful++
        }
        totalDuration += result.Metrics.Duration
    }

    log.Printf("Workflow Summary:")
    log.Printf("  Total steps: %d", len(results))
    log.Printf("  Successful: %d", successful)
    log.Printf("  Failed: %d", len(results)-successful)
    log.Printf("  Total duration: %v", totalDuration)
}
```

These examples demonstrate the flexibility and power of the pluggable work execution system, showing how different types of work can be executed using various backends while maintaining a consistent interface and workflow structure.