Pluggable Work Execution System API Documentation

Overview

The UBOM Workflow Engine's pluggable work execution system provides a flexible framework for executing different types of work using various execution backends. This document describes the API interfaces and usage patterns.

Core Interfaces

EnhancedWorkExecutor

The main interface for work executors with enhanced capabilities:

```
type EnhancedWorkExecutor interface {
    // Core execution methods
    Execute(ctx context.Context, work layer0.Work, workContext *layer0.Context) (WorkResult, error)
    Validate(work layer0.Work) error

    // Schema and metadata support
    GetSchema() WorkSchema
    GetMetadata() WorkMetadata

    // Compatibility with existing interface
    CanExecute(workType layer0.WorkType) bool
    GetSupportedTypes() []layer0.WorkType
}
```

WorkResult

Enhanced execution result with detailed information:

```
type WorkResult struct {
    Success bool                   `json:"success"`
    Outputs map[string]interface{} `json:"outputs"`
    Logs    []LogEntry             `json:"logs"`
    Metrics ExecutionMetrics       `json:"metrics"`
    Error   string                 `json:"error,omitempty"`
}
```

ExternalWorkPlugin

Interface for external plugins extending the base executor:

```go
type ExternalWorkPlugin interface {
    EnhancedWorkExecutor

    // Plugin lifecycle methods
    Initialize(config map[string]interface{}) error
    Shutdown() error
    HealthCheck() error

    // Plugin information
    GetPluginInfo() PluginInfo
}
```

Built-in Executors

Docker Executor

Executes work in Docker containers.

Work Type: `docker`

Configuration Example:

```json
{
  "executor_config": {
    "image": "ubuntu:20.04",
    "command": ["python", "script.py"],
    "environment": {
      "ENV_VAR": "value"
    },
    "volumes": [
      {
        "source": "/host/path",
        "target": "/container/path",
        "read_only": true
      }
    ],
    "resources": {
      "cpu_limit": "1.0",
      "memory_limit": "512m"
    }
  }
}
```

gRPC Executor

Makes remote procedure calls to gRPC services.

Work Type: `grpc`

Configuration Example:

```
{
  "executor_config": {
    "endpoint": "api.example.com:443",
    "method": "myservice.MyService/ProcessData",
    "tls": {
      "enabled": true
    },
    "metadata": {
      "authorization": "Bearer token123"
    },
    "retry": {
      "max_attempts": 3,
      "initial_delay": "1s"
    }
  }
}
```

Serverless Executor

Invokes cloud functions on AWS Lambda, Google Cloud Functions, or Azure Functions.

Work Type: `serverless`

Configuration Example:

```
{
  "executor_config": {
    "provider": "aws",
    "function": "my-lambda-function",
    "region": "us-east-1",
    "credentials": {
      "type": "aws_iam",
      "access_key": "AKIAIOSFODNN7EXAMPLE",
      "secret_key": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
    },
    "timeout": "30s"
  }
}
```

Registry Management

EnhancedWorkRegistry

Manages executor registration and discovery:

```
// Create registry
registry := executors.NewEnhancedWorkRegistry()

// Register executor
err := registry.RegisterExecutor(workType, executor)

// Get executor
executor, err := registry.GetExecutor(workType)

// Validate work
err := registry.ValidateWork(work)
```

Plugin System

Plugin Loader

Manages plugin lifecycle:

```
// Create loader
loader := plugins.NewDefaultPluginLoader(logger)

// Register plugin
factory := func() plugins.ExternalWorkPlugin {
    return NewMyPlugin()
}
err := loader.RegisterPlugin("my-plugin", factory)

// Initialize plugin
config := map[string]interface{}{"key": "value"}
err := loader.InitializePlugin("my-plugin", config)

// Get plugin
plugin, err := loader.GetPlugin("my-plugin")
```

Plugin Development

To create a custom plugin:

1. Implement the `ExternalWorkPlugin` interface
2. Extend `BasePlugin` for common functionality
3. Provide JSON schema for validation
4. Export a `NewPlugin` function for dynamic loading

Example plugin structure:

```
type MyPlugin struct {
    *plugins.BasePlugin
}

func NewPlugin() plugins.ExternalWorkPlugin {
    return &MyPlugin{
        BasePlugin: plugins.NewBasePlugin(info, metadata, schema),
    }
}

func (p *MyPlugin) Execute(ctx context.Context, work layer0.Work, workContext *lay-
er0.Context) (executors.WorkResult, error) {
    // Implementation
}
```

Schema Validation

SchemaValidator

Validates work configurations against JSON schemas:

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

// Register schema
validator.RegisterSchema(workType, schema)

// Validate work
err := validator.ValidateWork(work)
```

Error Handling

PluginError

Specialized error type for plugin operations:

```
type PluginError struct {
    PluginName string
    Operation  string
    Err        error
}
```

Backward Compatibility

BackwardCompatibilityAdapter

Adapts enhanced executors to work with the original interface:

```
enhanced := NewMyEnhancedExecutor()
adapter := executors.NewBackwardCompatibilityAdapter(enhanced)

// Use with original WorkExecutionCore
core := layer1.NewWorkExecutionCore()
err := core.RegisterExecutor(workType, adapter)
```

Best Practices

1. **Error Handling**: Always check for errors and provide meaningful error messages
2. **Resource Management**: Properly clean up resources in plugin shutdown methods
3. **Validation**: Validate all input configurations before execution
4. **Logging**: Use structured logging for better observability
5. **Testing**: Write comprehensive tests including unit, integration, and performance tests
6. **Documentation**: Provide clear documentation and examples for custom plugins

Performance Considerations

1. **Connection Pooling**: Reuse connections for network-based executors
2. **Resource Limits**: Set appropriate resource limits for containerized execution
3. **Timeouts**: Configure reasonable timeouts for all operations
4. **Concurrency**: Design executors to handle concurrent execution safely
5. **Metrics**: Collect and monitor execution metrics for performance optimization