Maggie AI Assistant: Logging and Error Handling System

Table of Contents

System Overview

Maggie AI Assistant implements comprehensive logging and error handling systems optimized for the AMD Ryzen 9 5900X and NVIDIA RTX 3080 hardware. These systems provide robust tracking, debugging, and recovery capabilities essential for a dependable AI assistant application.

Logging System Architecture

The logging system is built around a hierarchical architecture with the following key components:

- **LoggingManager**: Central manager implementing the Singleton pattern to ensure consistent logging throughout the application.
- **ComponentLogger**: Component-specific loggers that provide simplified interfaces for individual modules.
- **Contextual logging**: Correlation tracking and structured logging for tracing related log entries.
- **Performance tracking**: Automated metrics collection for monitoring system performance.
- **Multiple output destinations**: Console, file, and event bus publication options.

The logging system supports five severity levels defined in the `LogLevel` enum:

- `DEBUG`: Detailed information useful for debugging
- `INFO`: Confirmation that things are working as expected
- `WARNING`: Indication that something unexpected happened
- `ERROR`: More serious problem that prevented an operation from completing
- `CRITICAL`: Severe error that might require immediate attention

Error Handling System Architecture

The error handling system is structured around:

- **Custom exception hierarchy**: A tree of specialized exceptions starting with `MaggieError` as the base class.
- **ErrorContext**: A container for detailed error information including stack traces, state information, and correlation IDs.
- **Categorization**: Errors are categorized by domain (`ErrorCategory`) and severity (`ErrorSeverity`).
- **Safe execution patterns**: Utilities for executing code with proper error handling.
- **Retry mechanisms**: Support for automatic retries with configurable backoff strategies.

The system provides several specialized exception types:

- `MaggieError`: Base class for all custom exceptions
- `LLMError`: For language model issues
  - `ModelLoadError`: Specific to model loading failures
  - `GenerationError`: Specific to text generation failures
- `STTError`: For speech-to-text issues
- `TTSError`: For text-to-speech issues
- `ExtensionError`: For extension module issues
- `StateTransitionError`: For state machine transition failures
- `ResourceManagementError`: For resource allocation issues
- `InputProcessingError`: For user input processing failures

Integration Between Systems

The logging and error handling systems are tightly integrated:

1. Errors recorded through the error handling system are automatically logged with appropriate severity.
2. Error events are published to the event bus, allowing system-wide reaction to errors.
3. Both systems support correlation IDs for tracking related operations across components.
4. State information is captured in both logs and error contexts for easier debugging.
5. The systems integrate with the application's finite state machine for state-aware behavior.

Implementation Guide

This section provides detailed guidance on implementing logging and error handling in your Maggie AI components.

Setting Up Logging

Initializing the Logging System

The logging system is typically initialized during application startup in `main.py` through the `initialize_components` function:

```
from maggie.utils.logging import LoggingManager

# Configuration dict contains logging settings
config = {
    'logging': {
        'path': 'logs',
        'console_level': 'INFO',
        'file_level': 'DEBUG'
    }
}

# Initialize the logging manager - this should be done once during application startup
logging_manager = LoggingManager.initialize(config)
```

Once initialized, the LoggingManager can be accessed throughout the application using its singleton instance:

```
from maggie.utils.logging import LoggingManager

# Get the singleton instance
logging_manager = LoggingManager.get_instance()

# Use the manager to log messages
logging_manager.log(LogLevel.INFO, "Application started successfully")
```

Enhancing the Logging System

For full functionality, the logging system should be enhanced with event publishing, error handling, and state information:

```
# After initializing LoggingManager, enhance it with other systems
from maggie.utils.adapters import EventBusAdapter, StateManagerAdapter, ErrorHandlerAdapter

# Assuming you have already initialized event_bus, state_manager, error_handler
event_adapter = EventBusAdapter(event_bus)
state_adapter = StateManagerAdapter(state_manager)
error_adapter = ErrorHandlerAdapter()

# Enhance the logging manager
logging_manager.enhance_with_event_publisher(event_adapter)
logging_manager.enhance_with_state_provider(state_adapter)
```

```
logging_manager.enhance_with_error_handler(error_adapter)
```

Component-Specific Logging

For individual components, the recommended approach is to use the `ComponentLogger` class, which provides a simplified interface with component context:

```
from maggie.utils.logging import ComponentLogger

class MyComponent:
    def __init__(self):
        # Create a logger for this component
        self.logger = ComponentLogger('MyComponent')

    def do_something(self):
        # Log at different levels
        self.logger.debug("Detailed debug information")
        self.logger.info("Component initialized")
        self.logger.warning("Unusual condition detected")

        try:
            # Some operation that might fail
            result = self.risky_operation()
            return result
        except Exception as e:
            # Log errors with exception details
            self.logger.error("Failed to perform operation", exception=e)
            return None
```

Error Handling Implementation

Basic Error Recording

To record errors properly, use the `record_error` function:

```
from maggie.utils.error_handling import record_error, ErrorCategory, ErrorSeverity

try:
    # Some operation that might fail
    result = perform_operation()
except Exception as e:
    # Record the error with contextual information
    error_context = record_error(
        message="Failed to perform operation",
        exception=e,
        category=ErrorCategory.PROCESSING,
        severity=ErrorSeverity.ERROR,
        source="MyComponent.perform_operation",
        details={"operation_params": params, "state": current_state}
    )

    # The error is automatically logged and published to the event bus
```

```
    # error_context contains structured information about the error
```

Safe Execution

For operations that should be protected with automatic error handling, use the `safe_execute` function:

```
from maggie.utils.error_handling import safe_execute, ErrorCategory, ErrorSeverity

# Execute a function with error handling
result = safe_execute(
    risky_function,  # The function to execute
    arg1, arg2,      # Arguments to pass to the function
    error_code="OPERATION_FAILED",
    default_return=None,  # Value to return if an exception occurs
    error_category=ErrorCategory.PROCESSING,
    error_severity=ErrorSeverity.WARNING,
    publish_error=True,
    include_state_info=True
)
```

Using Decorators

For repeated error handling patterns, decorators provide a cleaner approach:

```
from maggie.utils.error_handling import with_error_handling, ErrorCategory, ErrorSeverity

class MyComponent:
    # Apply error handling to a method
    @with_error_handling(
        error_category=ErrorCategory.PROCESSING,
        error_severity=ErrorSeverity.ERROR
    )
    def process_data(self, data):
        # This method is now wrapped with error handling
        # If an exception occurs, it will be properly recorded and logged
        return transform_data(data)
```

Retry Logic

For operations that may fail transiently, use the retry decorator:

```
from maggie.utils.error_handling import retry_operation

class NetworkComponent:
    # Automatically retry network operations
    @retry_operation(
        max_attempts=3,
        retry_delay=1.0,
        exponential_backoff=True,
        jitter=True,
        allowed_exceptions=(ConnectionError, TimeoutError),
        error_category=ErrorCategory.NETWORK
    )
```

```
    def fetch_data(self, url):
        # This function will be retried up to 3 times if it raises
        # ConnectionError or TimeoutError
        response = requests.get(url, timeout=5)
        response.raise_for_status()
        return response.json()
```

Configuration Options

The logging and error handling systems can be configured through the application's configuration file (`config.yaml`). Key configuration options include:

Logging Configuration

```
logging:
  # Path to log directory
  path: "logs"

  # Log levels for console and file logging
  console_level: "INFO"
  file_level: "DEBUG"

  # Batch size for asynchronous logging
  batch_size: 50

  # Timeout for batched logging in seconds
  batch_timeout: 5.0

  # Enable/disable asynchronous logging
  async_enabled: true
```

Error Handling Configuration

Error handling is primarily configured through code rather than the configuration file, but it respects the logging configuration for error logging.

Code Examples

Basic Logging Usage

Example 1: Simple Component Logging

```
from maggie.utils.logging import ComponentLogger

class UserService:
    def __init__(self):
        self.logger = ComponentLogger("UserService")
        self.logger.info("UserService initialized")
```

```
    def authenticate_user(self, username, password):
        self.logger.debug(f"Authentication attempt for user: {username}")

        if not username or not password:
            self.logger.warning("Authentication attempt with empty credentials")
            return False

        try:
            # Authentication logic here
            is_authenticated = check_credentials(username, password)

            if is_authenticated:
                self.logger.info(f"User {username} authenticated successfully")
            else:
                self.logger.warning(f"Failed authentication attempt for user: {username}")

            return is_authenticated
        except Exception as e:
            self.logger.error(f"Authentication error for user: {username}", exception=e)
            return False
```

Example 2: Using Correlation IDs for Request Tracking

```
from maggie.utils.logging import LoggingManager, ComponentLogger
import uuid

class RequestHandler:
    def __init__(self):
        self.logger = ComponentLogger("RequestHandler")
        self.logging_manager = LoggingManager.get_instance()

    def handle_request(self, request_data):
        # Generate a correlation ID for tracking this request through the system
        correlation_id = str(uuid.uuid4())
        self.logging_manager.set_correlation_id(correlation_id)

        try:
            self.logger.info(f"Handling request", request_id=correlation_id)

            # Process the request
            response = self.process_request(request_data)

            self.logger.info(f"Request processed successfully",
                             request_id=correlation_id,
                             response_status="success")
            return response
        except Exception as e:
            self.logger.error(f"Error processing request",
                              exception=e,
                              request_id=correlation_id,
                              request_data=request_data)
            return {"error": "Failed to process request"}
        finally:
            # Clear the correlation ID
            self.logging_manager.clear_correlation_id()
```

Advanced Logging Techniques

Example 3: Logging Context for Operation Tracking

```python
from maggie.utils.logging import logging_context, ComponentLogger

class DataProcessor:
    def __init__(self):
        self.logger = ComponentLogger("DataProcessor")

    def process_batch(self, batch_id, items):
        # Create a logging context with batch_id as correlation ID
        with logging_context(
            correlation_id=batch_id,
            component="DataProcessor",
            operation="process_batch"
        ) as ctx:
            self.logger.info(f"Processing batch with {len(items)} items")

            # Update context with progress
            ctx["items_count"] = len(items)
            ctx["current_step"] = "validation"

            # Validate items
            valid_items = self.validate_items(items)

            # Update context progress
            ctx["valid_items"] = len(valid_items)
            ctx["current_step"] = "transformation"

            # Transform items
            transformed_items = self.transform_items(valid_items)

            # Update context again
            ctx["transformed_items"] = len(transformed_items)
            ctx["current_step"] = "storage"

            # Store items
            stored_count = self.store_items(transformed_items)

            # Final context update
            ctx["stored_items"] = stored_count
            ctx["success_rate"] = stored_count / len(items) if len(items) > 0 else 0

            return stored_count
```

Example 4: Performance Logging with the Decorator

```python
from maggie.utils.logging import log_operation, ComponentLogger

class QueryProcessor:
    def __init__(self):
        self.logger = ComponentLogger("QueryProcessor")

    @log_operation(component="QueryProcessor", log_args=True, log_result=False)
    def execute_query(self, query, params=None):
```

```
    """
    Execute a database query with automatic performance logging.

    The log_operation decorator will automatically:
    1. Log when the method is called with query and params
    2. Time the execution
    3. Log performance metrics when the method completes
    4. Include the execution time in the logs
    """
    # Query execution logic
    connection = self.get_connection()
    cursor = connection.cursor()

    try:
        cursor.execute(query, params or {})
        results = cursor.fetchall()
        return results
    finally:
        cursor.close()
        connection.close()
```

Error Handling Examples

Example 5: Basic Error Handling

```
from maggie.utils.error_handling import record_error, ErrorCategory, ErrorSeverity
from maggie.utils.logging import ComponentLogger

class FileManager:
    def __init__(self):
        self.logger = ComponentLogger("FileManager")

    def read_file(self, file_path):
        try:
            with open(file_path, 'r') as file:
                return file.read()
        except FileNotFoundError as e:
            # Record a specific error for file not found
            record_error(
                message=f"File not found: {file_path}",
                exception=e,
                category=ErrorCategory.INPUT,
                severity=ErrorSeverity.WARNING,
                source="FileManager.read_file",
                details={"file_path": file_path}
            )
            return None
        except PermissionError as e:
            # Record a different error for permission issues
            record_error(
                message=f"Permission denied for file: {file_path}",
                exception=e,
                category=ErrorCategory.PERMISSION,
                severity=ErrorSeverity.ERROR,
                source="FileManager.read_file",
                details={"file_path": file_path}
            )
```

```
            return None
        except Exception as e:
            # Record a generic error for other exceptions
            record_error(
                message=f"Error reading file: {file_path}",
                exception=e,
                category=ErrorCategory.UNKNOWN,
                severity=ErrorSeverity.ERROR,
                source="FileManager.read_file",
                details={"file_path": file_path}
            )
            return None
```

Example 6: Safe Execution Pattern

```
from maggie.utils.error_handling import safe_execute, ErrorCategory, ErrorSeverity
from maggie.utils.logging import ComponentLogger

class ModelManager:
    def __init__(self):
        self.logger = ComponentLogger("ModelManager")

    def load_model(self, model_path):
        # Use safe_execute to handle errors
        return safe_execute(
            self._load_model_internal,
            model_path,
            error_code="MODEL_LOAD_ERROR",
            default_return=None,
            error_category=ErrorCategory.MODEL,
            error_severity=ErrorSeverity.ERROR,
            error_details={"model_path": model_path},
            publish_error=True
        )

    def _load_model_internal(self, model_path):
        # This is the actual implementation that might raise exceptions
        self.logger.info(f"Loading model from {model_path}")

        if not os.path.exists(model_path):
            raise FileNotFoundError(f"Model file not found: {model_path}")

        # Model loading code that might raise various exceptions
        model = load_model_from_path(model_path)
        self.logger.info(f"Model loaded successfully: {model.name}")
        return model
```

Example 7: Decorator-Based Error Handling

```
from maggie.utils.error_handling import with_error_handling, ErrorCategory, ErrorSeverity
from maggie.utils.logging import ComponentLogger

class APIClient:
    def __init__(self, base_url):
        self.base_url = base_url
```

```
        self.logger = ComponentLogger("APIClient")

    @with_error_handling(
        error_code="API_REQUEST_FAILED",
        error_category=ErrorCategory.NETWORK,
        error_severity=ErrorSeverity.WARNING
    )
    def get_data(self, endpoint, params=None):
        """
        Get data from API endpoint.

        The with_error_handling decorator automatically:
        1. Executes the function in a try-except block
        2. Records any exceptions with appropriate context
        3. Returns None if an exception occurs
        """
        url = f"{self.base_url}/{endpoint}"
        self.logger.debug(f"Making API request to {url}")

        response = requests.get(url, params=params, timeout=10)
        response.raise_for_status()  # This might raise exceptions

        return response.json()
```

Example 8: Retry Logic for Network Operations

```
from maggie.utils.error_handling import retry_operation, ErrorCategory
from maggie.utils.logging import ComponentLogger
import requests

class ExternalServiceClient:
    def __init__(self, service_url):
        self.service_url = service_url
        self.logger = ComponentLogger("ExternalServiceClient")

    def log_retry(self, exception, attempt):
        self.logger.warning(
            f"Retry attempt {attempt} after error: {exception}",
            service_url=self.service_url
        )

    @retry_operation(
        max_attempts=3,
        retry_delay=2.0,
        exponential_backoff=True,
        jitter=True,
        allowed_exceptions=(requests.RequestException,),
        on_retry_callback=log_retry,
        error_category=ErrorCategory.NETWORK
    )
    def fetch_data(self, endpoint):
        """
        Fetch data from external service with automatic retries.

        This method will automatically retry up to 3 times with
        exponential backoff and jitter if a RequestException occurs.
        """
```

```
        url = f"{self.service_url}/{endpoint}"
        self.logger.debug(f"Fetching data from {url}")

        response = requests.get(url, timeout=5)
        response.raise_for_status()

        return response.json()
```

Combined Logging and Error Handling

Example 9: Advanced Component with Comprehensive Logging and Error Handling

```
from maggie.utils.logging import ComponentLogger, log_operation, logging_context
from maggie.utils.error_handling import (
    with_error_handling, retry_operation, safe_execute,
    ErrorCategory, ErrorSeverity, MaggieError
)
import time

class DataSynchronizer:
    def __init__(self, source_system, target_system):
        self.source_system = source_system
        self.target_system = target_system
        self.logger = ComponentLogger("DataSynchronizer")

    @log_operation(component="DataSynchronizer", log_args=True)
    def synchronize_data(self, data_type, since_timestamp=None):
        """
        Synchronize data between source and target systems.

        This high-level method demonstrates comprehensive logging and error handling:
        - Operation logging with the log_operation decorator
        - Contextual logging with the logging_context context manager
        - Safe execution pattern for error handling
        - Retry logic for transient failures
        """
        sync_id = f"sync-{data_type}-{int(time.time())}"

        with logging_context(
            correlation_id=sync_id,
            component="DataSynchronizer",
            operation=f"sync_{data_type}"
        ) as ctx:
            self.logger.info(
                f"Starting synchronization of {data_type}",
                source=self.source_system,
                target=self.target_system,
                since=since_timestamp
            )

            try:
                # Step 1: Fetch data from source
                ctx["step"] = "fetch_source_data"
                source_data = self._fetch_source_data(data_type, since_timestamp)
                ctx["records_fetched"] = len(source_data)

                if not source_data:
```

```
            self.logger.info(f"No {data_type} data to synchronize")
            return {"synchronized": 0, "errors": 0}

        # Step 2: Transform data
        ctx["step"] = "transform_data"
        transformed_data = self._transform_data(source_data, data_type)
        ctx["records_transformed"] = len(transformed_data)

        # Step 3: Upload to target
        ctx["step"] = "upload_to_target"
        result = self._upload_to_target(transformed_data, data_type)
        ctx["records_uploaded"] = result.get("successful", 0)
        ctx["upload_errors"] = result.get("errors", 0)

        # Step 4: Verify synchronization
        ctx["step"] = "verify_synchronization"
        verification_result = self._verify_synchronization(
            source_data, data_type
        )
        ctx["verified_records"] = verification_result.get("verified", 0)
        ctx["verification_errors"] = verification_result.get("errors", 0)

        # Final result
        sync_result = {
            "synchronized": result.get("successful", 0),
            "errors": result.get("errors", 0) + verification_result.get("errors", 0),
            "verification": verification_result
        }

        self.logger.info(
            f"Completed synchronization of {data_type}",
            result=sync_result
        )

        return sync_result

    except Exception as e:
        self.logger.error(
            f"Synchronization failed for {data_type}",
            exception=e,
            source=self.source_system,
            target=self.target_system
        )
        return {"synchronized": 0, "errors": 1, "error_message": str(e)}

@with_error_handling(
    error_category=ErrorCategory.NETWORK,
    error_severity=ErrorSeverity.ERROR
)
def _fetch_source_data(self, data_type, since_timestamp):
    """Fetch data from source system with error handling."""
    self.logger.debug(f"Fetching {data_type} data from {self.source_system}")
    # Implementation here
    return []

@with_error_handling(
    error_category=ErrorCategory.PROCESSING,
    error_severity=ErrorSeverity.ERROR
)
```

```
    def _transform_data(self, data, data_type):
        """Transform data with error handling."""
        self.logger.debug(f"Transforming {len(data)} {data_type} records")
        # Implementation here
        return []

    @retry_operation(
        max_attempts=3,
        retry_delay=2.0,
        exponential_backoff=True,
        jitter=True,
        error_category=ErrorCategory.NETWORK
    )
    def _upload_to_target(self, data, data_type):
        """Upload data to target system with retry logic."""
        self.logger.debug(f"Uploading {len(data)} {data_type} records to {self.target_system}")
        # Implementation here
        return {"successful": 0, "errors": 0}

    def _verify_synchronization(self, source_data, data_type):
        """Verify synchronization with safe execution."""
        return safe_execute(
            self._verify_synchronization_internal,
            source_data,
            data_type,
            error_category=ErrorCategory.PROCESSING,
            error_severity=ErrorSeverity.WARNING,
            default_return={"verified": 0, "errors": len(source_data)}
        )

    def _verify_synchronization_internal(self, source_data, data_type):
        """Internal verification implementation that might raise exceptions."""
        self.logger.debug(f"Verifying synchronization of {len(source_data)} {data_type} records")
        # Implementation here
        return {"verified": 0, "errors": 0}
```

Example 10: Error Propagation and Handling Across Components

```
from maggie.utils.logging import ComponentLogger
from maggie.utils.error_handling import (
    MaggieError, StateTransitionError, record_error,
    ErrorCategory, ErrorSeverity
)

# Custom exception for the workflow component
class WorkflowError(MaggieError):
    def __init__(self, message, workflow_id=None, step=None, details=None):
        self.workflow_id = workflow_id
        self.step = step
        self.details = details or {}
        super().__init__(message)

class WorkflowStep:
    def __init__(self, name, workflow):
        self.name = name
        self.workflow = workflow
        self.logger = ComponentLogger(f"WorkflowStep.{name}")
```

```
    def execute(self, context):
        """Execute this workflow step."""
        raise NotImplementedError("Subclasses must implement execute()")

    def handle_error(self, exception, context):
        """Default error handler for workflow steps."""
        if isinstance(exception, MaggieError):
            # Propagate existing Maggie errors with additional context
            if isinstance(exception, WorkflowError):
                # Already a workflow error, just add more context if needed
                if not exception.step:
                    exception.step = self.name
                if not exception.workflow_id:
                    exception.workflow_id = self.workflow.workflow_id
            else:
                # Wrap other Maggie errors in a WorkflowError
                raise WorkflowError(
                    f"Error in workflow step '{self.name}': {exception}",
                    workflow_id=self.workflow.workflow_id,
                    step=self.name,
                    details={"original_error": str(exception)}
                ) from exception
        else:
            # Convert other exceptions to WorkflowError
            raise WorkflowError(
                f"Error in workflow step '{self.name}': {exception}",
                workflow_id=self.workflow.workflow_id,
                step=self.name
            ) from exception

class Workflow:
    def __init__(self, workflow_id, steps=None):
        self.workflow_id = workflow_id
        self.steps = steps or []
        self.logger = ComponentLogger("Workflow")

    def add_step(self, step):
        """Add a step to the workflow."""
        self.steps.append(step)

    def execute(self, context=None):
        """Execute the entire workflow."""
        context = context or {}
        context['workflow_id'] = self.workflow_id

        self.logger.info(f"Starting workflow {self.workflow_id} with {len(self.steps)} steps")

        results = []

        for i, step in enumerate(self.steps):
            try:
                self.logger.info(f"Executing step {i+1}/{len(self.steps)}: {step.name}")
                result = step.execute(context)
                results.append(result)

                # Update context with step result
                context[f"step_{step.name}_result"] = result
```

```
            except WorkflowError as we:
                # Record the error and stop workflow
                record_error(
                    message=f"Workflow error in step {step.name}: {we}",
                    exception=we,
                    category=ErrorCategory.PROCESSING,
                    severity=ErrorSeverity.ERROR,
                    source=f"Workflow.{self.workflow_id}",
                    details={
                        "workflow_id": self.workflow_id,
                        "step": step.name,
                        "step_index": i,
                        "context": context,
                        **we.details
                    }
                )
                return {
                    "success": False,
                    "completed_steps": i,
                    "error_step": step.name,
                    "error": str(we),
                    "results": results
                }

            except Exception as e:
                # Record unexpected error
                record_error(
                    message=f"Unexpected error in workflow {self.workflow_id}, step {step.name}:
                    exception=e,
                    category=ErrorCategory.UNKNOWN,
                    severity=ErrorSeverity.ERROR,
                    source=f"Workflow.{self.workflow_id}",
                    details={
                        "workflow_id": self.workflow_id,
                        "step": step.name,
                        "step_index": i,
                        "context": context
                    }
                )
                return {
                    "success": False,
                    "completed_steps": i,
                    "error_step": step.name,
                    "error": str(e),
                    "results": results
                }

        self.logger.info(f"Workflow {self.workflow_id} completed successfully")
        return {
            "success": True,
            "completed_steps": len(self.steps),
            "results": results
        }

# Example usage:
class DataValidationStep(WorkflowStep):
    def execute(self, context):
        self.logger.info(f"Validating data for workflow {context['workflow_id']}")
```

```
        # Validation logic that might raise exceptions
        data = context.get('data')

        if not data:
            raise WorkflowError(
                "No data provided for validation",
                workflow_id=context['workflow_id'],
                step=self.name,
                details={"context_keys": list(context.keys())}
            )

        # Perform validation
        valid_records = []
        invalid_records = []

        for record in data:
            if self.validate_record(record):
                valid_records.append(record)
            else:
                invalid_records.append(record)

        return {
            "valid_count": len(valid_records),
            "invalid_count": len(invalid_records),
            "valid_records": valid_records,
            "invalid_records": invalid_records
        }

    def validate_record(self, record):
        # Validation logic
        return True
```

Best Practices

Logging Best Practices

1. **Use Component-Specific Loggers**: Create a dedicated `ComponentLogger` for each class or module.

2. **Choose Appropriate Log Levels**:

   - DEBUG: Detailed information for debugging
   - INFO: Confirmation that things are working as expected
   - WARNING: Something unexpected happened but not an error
   - ERROR: An operation failed
   - CRITICAL: System stability is at risk

3. **Include Contextual Information**: Always add relevant context to log messages:

```
logger.error("Database connection failed",
             server=db_server,
             port=db_port,
             retry_count=retry_count)
```

4. **Use Correlation IDs**: For operations that span multiple components, use correlation IDs to trace the flow.

5. **Log Performance Metrics**: For critical operations, log performance data:

```
logger.log_performance("process_query", elapsed_time,
                       {"query_size": size, "records_processed": count})
```

6. **Use Structured Logging**: Prefer structured logging over string formatting:

```
# Good
logger.info("User logged in", user_id=user.id, ip_address=request.ip)

# Avoid
logger.info(f"User {user.id} logged in from {request.ip}")
```

7. **Don't Log Sensitive Data**: Never log passwords, authentication tokens, personal information, etc.

8. **Use Context Managers for Complex Operations**:

```
with logging_context(operation="data_sync", component="Synchronizer") as ctx:
    # Operation steps with context updates
```

9. **Use Decorators for Repeated Patterns**:

```
@log_operation(component="DataService")
def process_data(data):
    # Implementation
```

Error Handling Best Practices

1. **Use the Custom Exception Hierarchy**: Use the appropriate exception type from the hierarchy or create your own subclass of `MaggieError`.

2. **Categorize Errors Properly**: Use the correct `ErrorCategory` to help with filtering and analysis.

3. **Set Appropriate Severity**: Use the correct `ErrorSeverity` based on the impact of the error.

4. **Include Detailed Context**: Always include relevant details when recording errors:

```
record_error(
    message="Failed to process file",
    exception=e,
    details={"file_path": file_path, "file_size": file_size}
)
```

5. **Use Safe Execution for Critical Operations**:

```
result = safe_execute(critical_function, *args, default_return=fallback_value)
```

6. **Add Retry Logic for Transient Failures**:

```
@retry_operation(max_attempts=3, exponential_backoff=True)
def fetch_remote_data():
    # Implementation that might have transient failures
```

7. **Handle Errors at the Appropriate Level**: Catch exceptions at the level where you can properly handle them, not necessarily where they occur.

8. **Fail Fast for Programming Errors**: For programming errors (like assertion failures), it's often better to let them propagate than to catch and handle them.

9. **Avoid Empty Catch Blocks**: Always do something meaningful when catching exceptions.

10. **Clean Up Resources in Finally Blocks**: Use `try` / `finally` or context managers to ensure resources are properly cleaned up.

Performance Considerations

1. **Asynchronous Logging**: For high-volume logging, enable asynchronous logging:

```
config = {
    'logging': {
        'async_enabled': True,
        'batch_size': 50,
        'batch_timeout': 5.0
    }
}
```

2. **Log Level Filtering**: Set appropriate log levels to reduce log volume:

```
config = {
    'logging': {
        'console_level': 'INFO',
        'file_level': 'DEBUG'
    }
}
```

3. **Avoid Expensive Computations in Log Statements**:

```
# Avoid - this will always compute the expression even if debug logs are disabled
logger.debug(f"User data: {get_user_details()}")

# Better - only computes if debug is enabled
if logger.isEnabledFor(logging.DEBUG):
    logger.debug(f"User data: {get_user_details()}")
```

4. **Be Selective with Performance Logging**: Only log performance metrics for significant operations.

5. **Monitor Log Volume**: Excessive logging can impact performance. Monitor log volume and adjust levels as needed.

6. **Use Batched Logging for High-Volume Operations**: Group related log entries using contexts or batch processing.

Reference Materials

API Reference

Logging System Classes

- **LoggingManager**
  - `initialize(config)` : Initialize the logging system
  - `get_instance()` : Get the singleton instance
  - `log(level, message, *args, exception=None, **kwargs)` : Log a message
  - `set_correlation_id(correlation_id)` : Set correlation ID
  - `get_correlation_id()` : Get current correlation ID
  - `clear_correlation_id()` : Clear correlation ID
  - `log_performance(component, operation, elapsed, details=None)` : Log performance metrics
  - `log_state_transition(from_state, to_state, trigger)` : Log state transition
  - `setup_global_exception_handler()` : Set up global exception handler
  - `enhance_with_event_publisher(event_publisher)` : Add event publishing
  - `enhance_with_error_handler(error_handler)` : Add error handling
  - `enhance_with_state_provider(state_provider)` : Add state awareness
- **ComponentLogger**
  - `__init__(component_name)` : Initialize component logger
  - `debug(message, **kwargs)` : Log debug message
  - `info(message, **kwargs)` : Log info message
  - `warning(message, **kwargs)` : Log warning message
  - `error(message, exception=None, **kwargs)` : Log error message
  - `critical(message, exception=None, **kwargs)` : Log critical message
  - `log_state_change(old_state, new_state, trigger)` : Log state change
  - `log_performance(operation, elapsed, details=None)` : Log performance metrics
- **Context Managers and Decorators**
  - `logging_context(correlation_id=None, component='', operation='', state=None)` : Context manager for structured logging
  - `log_operation(component='', log_args=True, log_result=False, include_state=True)` : Decorator for function logging

Error Handling System Functions

- **Core Functions**
  - `record_error(message, exception=None, category=ErrorCategory.UNKNOWN, severity=ErrorSeverity.ERROR, source='', details=None, publish=True, state_object=None, from_state=None, to_state=None, trigger=None)` : Record an error

- `safe_execute(func, *args, error_code=None, default_return=None, error_details=None, error_category=ErrorCategory.UNKNOWN, error_severity=ErrorSeverity.ERROR, publish_error=True, include_state_info=True, **kwargs)` : Safely execute a function
- `create_state_transition_error(from_state, to_state, trigger, details=None)` : Create a state transition error

- **Decorators**

  - `with_error_handling(error_code=None, error_category=ErrorCategory.UNKNOWN, error_severity=ErrorSeverity.ERROR, publish_error=True, include_state_info=True)` : Error handling decorator
  - `retry_operation(max_attempts=3, retry_delay=1.0, exponential_backoff=True, jitter=True, allowed_exceptions=(Exception,), on_retry_callback=None, error_category=ErrorCategory.UNKNOWN)` : Retry decorator

- **Exception Classes**

  - `MaggieError` : Base exception class
  - `LLMError` : Language model error
  - `ModelLoadError` : Model loading error
  - `GenerationError` : Text generation error
  - `STTError` : Speech-to-text error
  - `TTSError` : Text-to-speech error
  - `ExtensionError` : Extension module error
  - `StateTransitionError` : State transition error
  - `ResourceManagementError` : Resource management error
  - `InputProcessingError` : Input processing error

- **Error Context**

  - `ErrorContext` : Container for detailed error information
    - `__init__(message, exception=None, category=ErrorCategory.UNKNOWN, severity=ErrorSeverity.ERROR, source='', details=None, correlation_id=None, state_info=None)` : Initialize error context
    - `to_dict()` : Convert to dictionary
    - `log(publish=True)` : Log the error

Configuration Reference

Logging Configuration Options

Option

Type

Default

Description

`path`

str

“logs”

Directory for log files

`console_level`

str

“INFO”

Minimum level for console logging

`file_level`

str

“DEBUG”

Minimum level for file logging

`batch_size`

int

50

Maximum batch size for async logging

`batch_timeout`

float

5.0

Maximum wait time for batched logs

`async_enabled`

bool

True

Enable/disable async logging

Additional Resources

- Python’s logging module: https://docs.python.org/3/library/logging.html
- Exception handling: https://docs.python.org/3/tutorial/errors.html
- Context managers: https://docs.python.org/3/reference/datamodel.html#context-managers
- Decorators: https://docs.python.org/3/glossary.html#term-decorator
- State pattern: https://refactoring.guru/design-patterns/state
- Observer pattern: https://refactoring.guru/design-patterns/observer
- Command pattern: https://refactoring.guru/design-patterns/command
- Retry pattern: https://docs.microsoft.com/en-us/azure/architecture/patterns/retry

---

This manual provides comprehensive guidance for implementing the logging and error handling systems in the Maggie AI Assistant. By following these guidelines and examples, you can create robust, maintainable, and debuggable components that integrate well with the application's architecture.