MQTT Event Listener Documentation

Table of Contents

MQTT Event Listener is a Python library for MQTT event listening with comprehensive job tracking, configuration parsing, and error handling capabilities.

This is internal software for organizational use only. See the Internal Distribution Guide for installation and usage within your organization.

1. Quick Start

```
# Install from repository
pip install git+https://github.com/ed-00/Mqtt-client.git

# Import and use
python -c "from Listener import EventListener; print('Installation successful!')"
```

```
import asyncio
from Listener import EventListener, EventListenerConfig

async def main():
    config = EventListenerConfig(host="localhost", topic="events")
    listener = EventListener(config)

    def processor(data, job_id):
        print(f"Processing: {data}")
        return None

    await listener.run(processor)

asyncio.run(main())
```

2. Project Information

Version	1.0.0

License	Apache 2.0
Python	3.8+
Author	Abed Hameed
Organization	KTH Royal Institute of Technology
Repository	https://github.com/ed-00/Mqtt-client

3. Documentation Sections

4. Installation Guide

This section covers all installation methods for the MQTT Event Listener.

4.1. Prerequisites

Requirement	Description	Version
Python	Python programming language	3.8+
Git	Version control system	2.20+
Network Access	Access to internal repositories	Required

4.2. Installation Methods

4.2.1. Method 1: Direct from Repository (Recommended)

Install the latest version directly from the git repository:

```
# Latest version from main branch
pip install git+https://github.com/ed-00/Mqtt-client.git

# Specific version tag
pip install git+https://github.com/ed-00/Mqtt-client.git@v1.0.0

# From develop branch
pip install git+https://github.com/ed-00/Mqtt-client.git@develop
```

4.2.2. Method 2: Local Clone

For development or when you need the full repository:

```
# Clone repository
git clone https://github.com/ed-00/Mqtt-client.git
cd Mqtt-client
```

```
# Install package
pip install .

# Install in editable mode for development
pip install -e .[dev]
```

4.2.3. Method 3: Wheel Distribution

If you have a wheel file distributed internally:

```
# Install from wheel file
pip install mqtt-event-listener-1.0.0-py3-none-any.whl
```

4.3. Verification

Verify the installation was successful:

```
from Listener import EventListener, EventListenerConfig

# Check version
print(f"Installed version: {{EventListener.__version__}}")

# Quick functionality test
config = EventListenerConfig()
print("✅ Installation successful!")
```

4.4. Development Installation

For contributors and developers:

```
# Clone repository
git clone https://github.com/ed-00/Mqtt-client.git
cd Mqtt-client

# Install development dependencies
pip install -e .[dev]

# Verify development setup
make test
```

Development installation includes:

- Testing frameworks (pytest, pytest-asyncio, pytest-cov)
- Code quality tools (flake8, bandit, safety)

- Documentation tools (if needed)

4.5. Requirements Integration

For projects depending on this library, add to your `requirements.txt`:

```
# Latest version
git+https://github.com/ed-00/Mqtt-client.git

# Specific version
git+https://github.com/ed-00/Mqtt-client.git@v1.0.0

# Development branch
git+https://github.com/ed-00/Mqtt-client.git@develop
```

4.6. Troubleshooting Installation

4.6.1. Common Issues

Git Authentication

Ensure you have access to the repository. Contact your administrator if you receive authentication errors.

Python Version

Verify you're using Python 3.8 or later:

```
python --version
```

Network Access

Ensure you can access the internal git repository. Test with:

```
git ls-remote https://github.com/ed-00/Mqtt-client.git
```

Dependency Conflicts

If you encounter dependency conflicts, consider using a virtual environment:

```
python -m venv mqtt-env
source mqtt-env/bin/activate  # On Windows: mqtt-env\Scripts\activate
pip install git+{link-repo}.git
```

4.6.2. Getting Help

If you encounter installation issues:

1. Check the issue tracker for known problems
2. Contact the maintainer: aahameed@kth.se
3. Review the Internal Distribution Guide

5. Configuration Guide

The MQTT Event Listener uses the `EventListenerConfig` class for comprehensive configuration management.

5.1. Basic Configuration

5.1.1. Minimal Setup

```
from Listener import EventListener, EventListenerConfig

# Minimal configuration
config = EventListenerConfig(
    host="localhost",
    port=1883,
    topic="events"
)
```

5.1.2. Complete Configuration

```
config = EventListenerConfig(
    # Connection Settings
    host="mqtt.example.com",
    port=8883,
    client_id="my-listener",
    username="mqtt_user",
    password="secure_password",
    uri="mqtts://mqtt.example.com:8883",

    # SSL/TLS Settings
    cafile="/path/to/ca.crt",
    capath="/path/to/ca/",
    cadata="-----BEGIN CERTIFICATE-----...",

    # MQTT Client Settings
    keep_alive=60,
    ping_delay=1,
    auto_reconnect=True,
    reconnect_retries=5,
    cleansession=False,

    # Topic Settings
```

```
    topic="events/+",
    qos=1,
    retain=False,
    error_topic="events/errors",
    log_topic="events/logs",
    results_topic="events/results",

    # Job Tracking Settings
    max_jobs_in_memory=10000,
    job_cleanup_interval=3600,
    allow_job_id_generation=True,
    duplicate_action="reprocess"
)
```

5.2. Configuration Categories

5.2.1. Connection Settings

Parameter	Description	Default	Example
`host`	MQTT broker hostname	`localhost`	`mqtt.example.com`
`port`	MQTT broker port	`1883`	`8883`
`client_id`	Unique client identifier	`event-listener`	`my-app-v1`
`username`	Authentication username	`test`	`mqtt_user`
`password`	Authentication password	`test`	`secure_pass`
`uri`	Complete MQTT URI	Generated	`mqtts://host:8883`

5.2.2. SSL/TLS Configuration

Parameter	Description	Default	Example
`cafile`	CA certificate file path	`None`	`/etc/ssl/ca.crt`
`capath`	CA certificate directory	`None`	`/etc/ssl/certs/`
`cadata`	CA certificate data	`None`	Certificate string

When using SSL/TLS, ensure:

- Certificate files are accessible
- Certificates are valid and not expired
- Hostname matches certificate CN/SAN

5.2.3. MQTT Client Settings

Parameter	Description	Default	Recommended
keep_alive	Keep-alive interval (seconds)	10	60
ping_delay	Ping delay (seconds)	1	1
auto_reconnect	Automatic reconnection	True	True
reconnect_retries	Reconnection attempts	2	5
reconnect_max_interval	Max reconnect interval	10	60
cleansession	Clean session flag	True	False

5.2.4. Topic Configuration

Parameter	Description	Default	Example
topic	Primary subscription topic	test	events/+
qos	Quality of Service level	0	1
retain	Retain message flag	False	True
error_topic	Error message topic	test/error	events/errors
log_topic	Log message topic	test/log	events/logs
results_topic	Result message topic	test/results	events/results

Topic Wildcards

The library supports MQTT topic wildcards:

- ` - Single level wildcard: `events//status
- - Multi level wildcard: events/

```
# Subscribe to all events from any device
config = EventListenerConfig(topic="devices/+/events")

# Subscribe to all messages under events hierarchy
config = EventListenerConfig(topic="events/#")
```

5.2.5. Job Tracking Settings

Parameter	Description	Default	Recommendation
max_jobs_in_memory	Maximum jobs in memory	5000	10000
job_cleanup_interval	Cleanup interval (seconds)	259200	3600
job_id_field	TOML field for job ID	job_id	job_id

Parameter	Description	Default	Recommendation
`allow_job_id_generation`	Auto-generate job IDs	`False`	`True`
`duplicate_action`	Duplicate job handling	`skip`	`reprocess`

Duplicate Action Options

Option	Behavior
`skip`	Ignore duplicate jobs silently
`reprocess`	Process duplicate jobs again
`error`	Raise error for duplicate jobs

5.3. Advanced Configuration

5.3.1. Will Messages

Configure last will and testament messages:

```
config = EventListenerConfig(
    will={
        "topic": "status/offline",
        "message": "Client disconnected",
        "qos": 1,
        "retain": True
    }
)
```

5.3.2. Custom Topics

Define multiple custom topics with individual settings:

```
config = EventListenerConfig(
    custom_topics={
        "alerts": {"qos": 2, "retain": True},
        "metrics": {"qos": 0, "retain": False},
        "commands": {"qos": 1, "retain": False}
    }
)
```

5.3.3. Configuration from Environment

Load configuration from environment variables:

```
import os

config = EventListenerConfig(
    host=os.getenv("MQTT_HOST", "localhost"),
    port=int(os.getenv("MQTT_PORT", 1883)),
    username=os.getenv("MQTT_USERNAME"),
    password=os.getenv("MQTT_PASSWORD"),
    topic=os.getenv("MQTT_TOPIC", "events")
)
```

5.3.4. Configuration from Files

```
import json

# Load from JSON
with open("config.json") as f:
    config_data = json.load(f)

config = EventListenerConfig(**config_data)
```

Example `config.json`:

```
{
    "host": "mqtt.example.com",
    "port": 8883,
    "username": "client",
    "password": "secret",
    "topic": "events/+",
    "auto_reconnect": true,
    "max_jobs_in_memory": 10000
}
```

5.4. Configuration Validation

The configuration class includes built-in validation:

```
try:
    config = EventListenerConfig(
        host="invalid-host",
        port=99999,  # Invalid port
        qos=5        # Invalid QoS
    )
except ValueError as e:
    print(f"Configuration error: {e}")
```

5.5. Best Practices

Configuration Best Practices:

1. **Use environment variables** for sensitive data (passwords, certificates)
2. **Set appropriate timeouts** for your network conditions
3. **Enable auto-reconnect** for production systems
4. **Use QoS 1 or 2** for critical messages
5. **Set meaningful client IDs** for debugging
6. **Configure will messages** for status monitoring
7. **Set appropriate job limits** based on memory constraints

6. Usage Guide

This section provides comprehensive usage examples for the MQTT Event Listener.

6.1. Basic Usage Pattern

6.1.1. Simple Event Listener

```
import asyncio
from Listener import EventListener, EventListenerConfig

async def main():
    # Configure the listener
    config = EventListenerConfig(
        host="localhost",
        port=1883,
        topic="events",
        client_id="my-listener"
    )

    # Create listener instance
    listener = EventListener(config)

    # Define message processing function
    def process_message(data, job_id):
        print(f"Processing job {job_id}: {data}")

        # Your processing logic here
        result = {"status": "processed", "job_id": job_id}

        # Return results to be published
        return ReturnType(
            data=result,
```

```
            topic="results",
            qos=0,
            retain=False,
            message_id=1,
            timestamp=datetime.now(),
            job_id=job_id
        )

    # Start listening
    await listener.run(process_message)

if __name__ == "__main__":
    asyncio.run(main())
```

6.2. Message Processing

6.2.1. TOML Message Format

The library expects messages in TOML format:

```
job_id = "task-001"
task_type = "data_processing"
priority = "high"
timestamp = "2025-01-09T10:30:00Z"

[data]
input_file = "/path/to/input.csv"
output_file = "/path/to/output.json"
parameters = { timeout = 300, retries = 3 }

[metadata]
user = "data_processor"
department = "analytics"
```

6.2.2. Processing Function Patterns

Basic Processing

```
def simple_processor(data, job_id):
    """Simple message processor."""
    print(f"Processing: {data.get('task_type')}")

    # Process the data
    result = {"job_id": job_id, "status": "completed"}

    return create_return_type(result, "results/simple")
```

Error Handling

```
def robust_processor(data, job_id):
    """Processor with error handling."""
    try:
        # Validate input
        if not data.get('task_type'):
            raise ValueError("Missing task_type")

        # Process data
        result = process_business_logic(data)

        return create_return_type(result, "results/success")

    except Exception as e:
        error_result = {
            "job_id": job_id,
            "error": str(e),
            "status": "failed"
        }
        return create_return_type(error_result, "results/errors")
```

Async Processing

```
async def async_processor(data, job_id):
    """Asynchronous message processor."""
    # Async operations
    async with aiohttp.ClientSession() as session:
        result = await external_api_call(session, data)

    # Database operations
    await save_to_database(result)

    return create_return_type(result, "results/async")

# Register async processor
await listener.run(async_processor)
```

6.3. Job Management

6.3.1. Monitoring Job Status

```
# Get specific job status
job_info = await listener.get_job_status("job-123")
if job_info:
    print(f"Job {job_info.job_id} status: {job_info.status}")
    print(f"Started: {job_info.started_at}")
```

```
    if job_info.completed_at:
        print(f"Completed: {job_info.completed_at}")

# Check if job is running
is_running = await listener.is_job_running("job-123")
print(f"Job running: {is_running}")

# Check if job exists
exists = await listener.job_exists("job-123")
print(f"Job exists: {exists}")
```

6.3.2. Querying Jobs by Status

```
# Get all running jobs
running_jobs = await listener.get_running_jobs()
print(f"Running jobs: {len(running_jobs)}")

# Get completed jobs
completed_jobs = await listener.get_completed_jobs()
for job_id, job_info in completed_jobs.items():
    print(f"Job {job_id}: {job_info.result}")

# Get duplicate jobs
duplicates = await listener.get_duplicate_jobs()
print(f"Duplicate jobs detected: {len(duplicates)}")

# Get all jobs
all_jobs = await listener.get_all_jobs()
print(f"Total jobs in memory: {len(all_jobs)}")
```

6.3.3. Job Cleanup

```
# Manual cleanup of old jobs
await listener.cleanup_old_jobs()

# Automatic cleanup is handled by the cleanup_interval setting
config = EventListenerConfig(
    job_cleanup_interval=3600  # Cleanup every hour
)
```

6.4. Configuration Patterns

6.4.1. Environment-Based Configuration

```
import os
```

```
def create_config_from_env():
    """Create configuration from environment variables."""
    return EventListenerConfig(
        host=os.getenv("MQTT_HOST", "localhost"),
        port=int(os.getenv("MQTT_PORT", 1883)),
        username=os.getenv("MQTT_USERNAME"),
        password=os.getenv("MQTT_PASSWORD"),
        topic=os.getenv("MQTT_TOPIC", "events"),
        client_id=os.getenv("MQTT_CLIENT_ID", "event-listener"),

        # SSL settings
        cafile=os.getenv("MQTT_CA_FILE"),

        # Job settings
        max_jobs_in_memory=int(os.getenv("MAX_JOBS", 5000)),
        job_cleanup_interval=int(os.getenv("CLEANUP_INTERVAL", 3600))
    )

config = create_config_from_env()
```

6.4.2. Multi-Environment Configuration

```
def get_config(environment="development"):
    """Get configuration for different environments."""
    configs = {
        "development": EventListenerConfig(
            host="localhost",
            port=1883,
            topic="dev/events",
            auto_reconnect=True,
            max_jobs_in_memory=1000
        ),

        "staging": EventListenerConfig(
            host="staging-mqtt.example.com",
            port=8883,
            topic="staging/events",
            auto_reconnect=True,
            reconnect_retries=5,
            max_jobs_in_memory=5000,
            cafile="/etc/ssl/staging-ca.crt"
        ),

        "production": EventListenerConfig(
            host="mqtt.example.com",
            port=8883,
            topic="events",
            auto_reconnect=True,
            reconnect_retries=10,
```

```
            max_jobs_in_memory=10000,
            cafile="/etc/ssl/production-ca.crt",
            job_cleanup_interval=1800
        )
    }

    return configs.get(environment, configs["development"])
```

6.5. Advanced Usage Patterns

6.5.1. Custom Configuration Parser

```
import logging
from Listener import SafeConfigParser

# Create custom parser
logger = logging.getLogger(__name__)
parser = SafeConfigParser(logger)

# Use with EventListener
listener = EventListener(config, config_parser=parser)
```

6.5.2. Multiple Topic Handling

```
def multi_topic_processor(data, job_id):
    """Process messages from different topics."""
    topic = data.get('_topic')  # Topic info from context

    if topic.startswith('alerts/'):
        return process_alert(data, job_id)
    elif topic.startswith('metrics/'):
        return process_metric(data, job_id)
    elif topic.startswith('commands/'):
        return process_command(data, job_id)
    else:
        return process_default(data, job_id)

# Configure for multiple topics
config = EventListenerConfig(
    topic="events/#",  # Subscribe to all under events
    custom_topics={
        "alerts/+": {"qos": 2, "retain": True},
        "metrics/+": {"qos": 0, "retain": False},
        "commands/+": {"qos": 1, "retain": False}
    }
)
```

6.5.3. Graceful Shutdown

```
import signal
import asyncio

class GracefulEventListener:
    def __init__(self, config):
        self.listener = EventListener(config)
        self.shutdown_event = asyncio.Event()

    def signal_handler(self, signum, frame):
        """Handle shutdown signals."""
        print(f"Received signal {signum}, shutting down...")
        self.listener.stop()
        self.shutdown_event.set()

    async def run(self, processor):
        """Run with graceful shutdown."""
        # Register signal handlers
        signal.signal(signal.SIGINT, self.signal_handler)
        signal.signal(signal.SIGTERM, self.signal_handler)

        try:
            # Start listener
            listener_task = asyncio.create_task(
                self.listener.run(processor)
            )

            # Wait for shutdown
            await self.shutdown_event.wait()

            # Wait for listener to stop
            await listener_task

        except Exception as e:
            print(f"Error during operation: {e}")
        finally:
            print("Shutdown complete")

# Usage
async def main():
    config = EventListenerConfig(host="localhost", topic="events")
    graceful_listener = GracefulEventListener(config)
    await graceful_listener.run(my_processor)
```

6.6. Performance Optimization

6.6.1. Batch Processing

```
class BatchProcessor:
    def __init__(self, batch_size=10, timeout=5.0):
        self.batch_size = batch_size
        self.timeout = timeout
        self.batch = []
        self.last_batch_time = time.time()

    async def process_message(self, data, job_id):
        """Add message to batch for processing."""
        self.batch.append((data, job_id))

        # Process batch if full or timeout reached
        if (len(self.batch) >= self.batch_size or
            time.time() - self.last_batch_time > self.timeout):
            await self.process_batch()

    async def process_batch(self):
        """Process accumulated batch."""
        if not self.batch:
            return

        print(f"Processing batch of {len(self.batch)} messages")

        # Process all messages in batch
        results = []
        for data, job_id in self.batch:
            result = await process_single_message(data, job_id)
            results.append(result)

        # Send batch results
        await send_batch_results(results)

        # Reset batch
        self.batch = []
        self.last_batch_time = time.time()

# Usage
batch_processor = BatchProcessor()
await listener.run(batch_processor.process_message)
```

6.6.2. Memory Management

```
# Configure for memory efficiency
config = EventListenerConfig(
    max_jobs_in_memory=1000,      # Limit job memory usage
    job_cleanup_interval=300,     # Cleanup every 5 minutes
    duplicate_action="skip"       # Avoid duplicate processing
```

```
)

# Monitor memory usage
import psutil

def monitor_memory():
    """Monitor memory usage."""
    process = psutil.Process()
    memory_mb = process.memory_info().rss / 1024 / 1024
    print(f"Memory usage: {memory_mb:.1f} MB")

# Periodic memory monitoring
async def memory_monitor():
    while True:
        monitor_memory()
        await asyncio.sleep(60)  # Check every minute
```

6.7. Error Handling Patterns

6.7.1. Comprehensive Error Handling

```
import logging
from Listener import ConfigError

logger = logging.getLogger(__name__)

async def robust_main():
    """Main function with comprehensive error handling."""
    try:
        # Configuration
        config = EventListenerConfig(
            host="mqtt.example.com",
            topic="events"
        )

        # Create listener
        listener = EventListener(config)

        # Define processor with error handling
        def error_tolerant_processor(data, job_id):
            try:
                return process_data_safely(data, job_id)
            except ValueError as e:
                logger.error(f"Data validation error for job {job_id}: {e}")
                return create_error_response(job_id, "validation_error", str(e))
            except Exception as e:
                logger.exception(f"Unexpected error for job {job_id}")
                return create_error_response(job_id, "processing_error", str(e))
```

```
        # Run listener
        await listener.run(error_tolerant_processor)

    except ConfigError as e:
        logger.error(f"Configuration error: {e}")
    except ConnectionError as e:
        logger.error(f"Connection error: {e}")
    except Exception as e:
        logger.exception("Unexpected error in main")
    finally:
        logger.info("Application shutdown")

def create_error_response(job_id, error_type, message):
    """Create standardized error response."""
    return ReturnType(
        data={
            "job_id": job_id,
            "status": "error",
            "error_type": error_type,
            "error_message": message,
            "timestamp": datetime.now().isoformat()
        },
        topic="errors",
        qos=1,
        retain=False,
        message_id=int(time.time()),
        timestamp=datetime.now(),
        job_id=job_id
    )
```

6.8. Best Practices

Usage Best Practices:

1. **Always handle exceptions** in your processing functions
2. **Use meaningful job IDs** for debugging and tracking
3. **Implement proper logging** for troubleshooting
4. **Monitor job queues** to prevent memory issues
5. **Use QoS appropriately** for message reliability
6. **Implement graceful shutdown** for production systems
7. **Test with various message formats** and edge cases
8. **Monitor performance metrics** regularly

7. API Reference

Complete API documentation for the MQTT Event Listener.

7.1. Core Classes

7.1.1. EventListener

The main class for MQTT event listening with job tracking.

```
class EventListener:
    """MQTT Event Listener with job tracking capabilities."""
```

Constructor

```
def __init__(self, config: EventListenerConfig,
             config_parser: Optional[SafeConfigParser] = None) -> None
```

Parameters:

- `config` (EventListenerConfig): Configuration object containing all settings
- `config_parser` (Optional[SafeConfigParser]): Custom config parser instance

Example:

```
config = EventListenerConfig(host="localhost", topic="events")
listener = EventListener(config)
```

Methods

run()

```
async def run(self, func: Callable[[Dict[str, Any], str],
              Optional[ReturnType]]) -> None
```

Start the event listener and process incoming messages.

Parameters:

- `func`: Message processing function that takes (data, job_id) and returns Optional[ReturnType]

Example:

```
def my_processor(data, job_id):
```

```
    return ReturnType(data=result, topic="results", ...)

await listener.run(my_processor)
```

stop()

```
def stop() -> None
```

Stop the event listener gracefully.

Example:

```
listener.stop()
```

Job Management Methods

```
async def get_job_status(self, job_id: str) -> Optional[JobInfo]
async def is_job_completed(self, job_id: str) -> bool
async def is_job_running(self, job_id: str) -> bool
async def job_exists(self, job_id: str) -> bool
async def get_all_jobs(self) -> Dict[str, JobInfo]
async def get_running_jobs(self) -> Dict[str, JobInfo]
async def get_completed_jobs(self) -> Dict[str, JobInfo]
async def get_duplicate_jobs(self) -> Dict[str, JobInfo]
async def cleanup_old_jobs(self) -> None
```

Job Status Methods:

Method	Description	Return Type
`get_job_status()`	Get detailed job information	`Optional[JobInfo]`
`is_job_completed()`	Check if job is completed	`bool`
`is_job_running()`	Check if job is currently running	`bool`
`job_exists()`	Check if job exists in memory	`bool`

Job Query Methods:

Method	Description	Return Type
`get_all_jobs()`	Get all jobs in memory	`Dict[str, JobInfo]`
`get_running_jobs()`	Get only running jobs	`Dict[str, JobInfo]`
`get_completed_jobs()`	Get only completed jobs	`Dict[str, JobInfo]`
`get_duplicate_jobs()`	Get jobs marked as duplicates	`Dict[str, JobInfo]`

Maintenance Methods:

Method	Description
`cleanup_old_jobs()`	Remove old completed jobs from memory

Examples:

```
# Check job status
job_info = await listener.get_job_status("job-123")
if job_info and job_info.status == JobStatus.COMPLETED:
    print(f"Job completed with result: {job_info.result}")

# Get running jobs count
running = await listener.get_running_jobs()
print(f"Currently running: {len(running)} jobs")

# Cleanup old jobs
await listener.cleanup_old_jobs()
```

7.1.2. EventListenerConfig

Configuration dataclass for the EventListener.

```
@dataclass(frozen=True)
class EventListenerConfig:
    """Unified configuration for the MQTT Event Listener."""
```

Configuration Parameters

Connection Settings

Parameter	Type	Default	Description
`host`	`str`	`"localhost"`	MQTT broker hostname
`port`	`int`	`1883`	MQTT broker port
`client_id`	`str`	`"event-listener"`	Unique client identifier
`username`	`str`	`"test"`	Authentication username
`password`	`str`	`"test"`	Authentication password
`uri`	`str`	`"mqtt://localhost:1883"`	Complete MQTT URI

SSL/TLS Settings

Parameter	Type	Default	Description
cafile	Optional[str]	None	CA certificate file path
capath	Optional[str]	None	CA certificate directory
cadata	Optional[str]	None	CA certificate data
additional_headers	Optional[Dict]	None	Additional HTTP headers

MQTT Client Settings

Parameter	Type	Default	Description
keep_alive	int	10	Keep-alive interval (seconds)
ping_delay	int	1	Ping delay (seconds)
default_qos	int	0	Default Quality of Service
default_retain	bool	False	Default retain flag
auto_reconnect	bool	True	Automatic reconnection
connect_timeout	Optional[int]	None	Connection timeout
reconnect_retries	int	2	Reconnection attempts
reconnect_max_interval	int	10	Max reconnect interval
cleansession	bool	True	Clean session flag

Topic Settings

Parameter	Type	Default	Description
topic	str	"test"	Primary subscription topic
qos	int	0	Quality of Service level
retain	bool	False	Retain message flag
error_topic	str	"test/error"	Error message topic
log_topic	str	"test/log"	Log message topic
results_topic	str	"test/results"	Result message topic

Advanced Settings

Parameter	Type	Default	Description
will	Optional[Dict]	None	Will message configuration
custom_topics	Dict[str, Dict]	{}	Custom topic configurations

Job Tracking Settings

Parameter	Type	Default	Description
`max_jobs_in_memory`	`int`	`5000`	Maximum jobs in memory
`job_cleanup_interval`	`int`	`259200`	Cleanup interval (seconds)
`job_id_field`	`str`	`"job_id"`	TOML field for job ID
`allow_job_id_generation`	`bool`	`False`	Auto-generate job IDs
`duplicate_action`	`str`	`"skip"`	Duplicate job handling

Example:

```
config = EventListenerConfig(
    host="mqtt.example.com",
    port=8883,
    topic="events/+",
    auto_reconnect=True,
    max_jobs_in_memory=10000,
    job_cleanup_interval=3600
)
```

7.1.3. JobInfo

Information about a job execution.

```
@dataclass
class JobInfo:
    """Information about a job run."""
```

Attributes

Attribute	Type	Description
`job_id`	`str`	Unique identifier for the job
`status`	`JobStatus`	Current status of the job
`started_at`	`datetime`	When the job was created
`completed_at`	`Optional[datetime]`	When the job completed
`input_data`	`Optional[Dict[str, Any]]`	Input data for the job
`result`	`Optional[Any]`	Result returned by job function
`error`	`Optional[str]`	Error message if job failed

Example:

```
job_info = await listener.get_job_status("job-123")
if job_info:
    print(f"Job ID: {job_info.job_id}")
    print(f"Status: {job_info.status}")
    print(f"Started: {job_info.started_at}")
    if job_info.result:
        print(f"Result: {job_info.result}")
```

7.1.4. ReturnType

Return type for processed messages.

```
@dataclass
class ReturnType:
    """Return type for processed messages."""
```

Attributes

Attribute	Type	Description
`data`	`Dict[str, Any]`	Processed data to publish
`topic`	`str`	MQTT topic for publication
`qos`	`int`	Quality of Service level
`retain`	`bool`	Retain message flag
`message_id`	`int`	Unique message identifier
`timestamp`	`datetime`	Processing timestamp
`job_id`	`str`	Associated job ID

Example:

```
def my_processor(data, job_id):
    result = {"status": "processed", "data": data}

    return ReturnType(
        data=result,
        topic="results/processed",
        qos=1,
        retain=False,
        message_id=int(time.time()),
        timestamp=datetime.now(),
        job_id=job_id
    )
```

7.1.5. SafeConfigParser

Safe TOML configuration parser with error handling.

```
class SafeConfigParser:
    """Safe TOML configuration parser."""
```

Constructor

```
def __init__(self, logger: Optional[logging.Logger] = None) -> None
```

Parameters:

- `logger` (Optional[logging.Logger]): Logger instance for error reporting

Methods

```
def parse_config_from_dict(self, config_dict: Dict[str, Any]) -> Dict[str, Any]
def parse_config_from_string(self, config_string: str) -> Dict[str, Any]
def parse_config_from_file(self, file_path: str) -> Dict[str, Any]
def validate_config(self, config: Dict[str, Any]) -> bool
```

Examples:

```
import logging
from Listener import SafeConfigParser

logger = logging.getLogger(__name__)
parser = SafeConfigParser(logger)

# Parse from string
config_data = parser.parse_config_from_string(toml_string)

# Parse from file
config_data = parser.parse_config_from_file("config.toml")

# Validate configuration
is_valid = parser.validate_config(config_data)
```

7.2. Enumerations

7.2.1. JobStatus

Job execution status enumeration.

```
class JobStatus(Enum):
    """Job execution status enumeration."""

    PENDING = "pending"
    RUNNING = "running"
    COMPLETED = "completed"
    FAILED = "failed"
    DUPLICATE = "duplicate"
```

Status Values

Status	Description
`PENDING`	Job is queued but not yet started
`RUNNING`	Job is currently being processed
`COMPLETED`	Job has completed successfully
`FAILED`	Job has failed with an error
`DUPLICATE`	Job was detected as a duplicate

Example:

```
from Listener import JobStatus

# Check job status
job_info = await listener.get_job_status("job-123")
if job_info.status == JobStatus.COMPLETED:
    print("Job completed successfully")
elif job_info.status == JobStatus.FAILED:
    print(f"Job failed: {job_info.error}")
```

7.3. Exceptions

7.3.1. ConfigError

Raised when configuration parsing or validation fails.

```
class ConfigError(Exception):
    """Configuration error exception."""
```

Example:

```
from Listener import ConfigError

try:
```

```
    parser = SafeConfigParser()
    config = parser.parse_config_from_file("invalid.toml")
except ConfigError as e:
    print(f"Configuration error: {e}")
```

7.4. Utility Functions

7.4.1. Helper Functions

These utility functions can help with common tasks:

```
def create_return_type(data: Dict[str, Any], topic: str,
                       job_id: str, qos: int = 0,
                       retain: bool = False) -> ReturnType:
    """Helper to create ReturnType instances."""
    return ReturnType(
        data=data,
        topic=topic,
        qos=qos,
        retain=retain,
        message_id=int(time.time()),
        timestamp=datetime.now(),
        job_id=job_id
    )

def create_error_result(job_id: str, error_message: str,
                        error_type: str = "processing_error") -> ReturnType:
    """Helper to create error result."""
    return create_return_type(
        data={
            "job_id": job_id,
            "status": "error",
            "error_type": error_type,
            "error_message": error_message,
            "timestamp": datetime.now().isoformat()
        },
        topic="errors",
        job_id=job_id,
        qos=1
    )
```

7.5. Message Processing Function Signature

Your message processing function must follow this signature:

```
def processor(data: Dict[str, Any], job_id: str) -> Optional[ReturnType]:
    """
```

```
    Process an incoming MQTT message.

    Args:
        data: Parsed TOML data as dictionary
        job_id: Unique job identifier

    Returns:
        Optional[ReturnType]: Result to publish, or None for no response
    """
    pass
```

Or for async processing:

```
async def async_processor(data: Dict[str, Any],
                          job_id: str) -> Optional[ReturnType]:
    """
    Process an incoming MQTT message asynchronously.

    Args:
        data: Parsed TOML data as dictionary
        job_id: Unique job identifier

    Returns:
        Optional[ReturnType]: Result to publish, or None for no response
    """
    pass
```

7.6. Type Hints

The library includes comprehensive type hints for better IDE support:

```
from typing import Dict, Any, Optional, Callable
from datetime import datetime
from Listener import EventListener, EventListenerConfig, ReturnType

# Function type for message processors
ProcessorFunc = Callable[[Dict[str, Any], str], Optional[ReturnType]]

# Example with proper typing
def typed_processor(data: Dict[str, Any], job_id: str) -> Optional[ReturnType]:
    # Type-safe processing
    task_type: str = data.get("task_type", "unknown")
    priority: int = data.get("priority", 1)

    result: Dict[str, Any] = {
        "job_id": job_id,
        "task_type": task_type,
        "status": "completed"
```

```
    }

    return ReturnType(
        data=result,
        topic="results",
        qos=0,
        retain=False,
        message_id=1,
        timestamp=datetime.now(),
        job_id=job_id
    )
```

8. Development Guide

Guide for contributors and developers working on the MQTT Event Listener.

8.1. Development Environment Setup

8.1.1. Prerequisites

Tool	Description	Version
Python	Python programming language	3.8+
Git	Version control system	2.20+
Make	Build automation tool	Any
Docker	Containerization (optional)	20+

8.1.2. Initial Setup

```
# Clone the repository
git clone https://github.com/ed-00/Mqtt-client.git
cd Mqtt-client

# Create virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install development dependencies
pip install -e .[dev]

# Verify setup
make test
```

8.1.3. Development Dependencies

The development installation includes:

Package	Purpose	Version
pytest	Testing framework	7.0.0+
pytest-asyncio	Async test support	0.21.0+
pytest-mock	Mocking utilities	3.10.0+
pytest-cov	Coverage reporting	4.0.0+
flake8	Code linting	6.0.0+
bandit	Security analysis	1.7.0+
safety	Dependency scanning	2.0.0+

8.2. Project Structure

8.2.1. Directory Layout

```
Mqtt-client/
 ├──── Listener/                    # Main package
 │    ├──── __init__.py             # Package initialization
 │    ├──── event_listener.py       # Core EventListener class
 │    ├──── safe_config_parser.py   # Configuration parser
 │    └──── py.typed                # Type hints marker
 ├──── tests/                      # Test suite
 │    ├──── __init__.py
 │    ├──── conftest.py             # Test configuration
 │    ├──── test_event_listener.py  # EventListener tests
 │    ├──── test_safe_config_parser.py # Parser tests
 │    └──── test_integration.py     # Integration tests
 ├──── docs/                       # Documentation
 │    ├──── asciidoc/               # AsciiDoc source
 │    ├──── html/                   # Generated HTML
 │    └──── pdf/                    # Generated PDF
 ├──── scripts/                    # Build and utility scripts
 │    ├──── build_package.sh        # Package build script
 │    └──── run_tests.py            # Test runner
 ├──── .github/                    # GitHub configuration
 │    └──── workflows/              # CI/CD workflows
 ├──── pyproject.toml              # Modern Python packaging
 ├──── MANIFEST.in                 # Package manifest
 ├──── README.md                   # Project readme
 ├──── LICENSE                     # License file
 ├──── requirements.txt            # Dependencies
 ├──── pytest.ini                 # Test configuration
 ├──── Makefile                    # Build automation
```

```
└──── .gitignore                 # Git ignore rules
```

8.2.2. Package Architecture

```
Listener Package:
├──── EventListener              # Main listener class
│    ├──── Job tracking           # In-memory job management
│    ├──── MQTT client            # AMQTT client wrapper
│    ├──── Message processing     # TOML message handling
│    └──── Error handling         # Exception management
├──── EventListenerConfig       # Configuration dataclass
│    ├──── Connection settings    # MQTT broker config
│    ├──── Job tracking config    # Job management settings
│    └──── SSL/TLS settings       # Security configuration
├──── SafeConfigParser          # TOML parser with validation
│    ├──── String parsing         # Parse TOML strings
│    ├──── File parsing           # Parse TOML files
│    └──── Validation             # Configuration validation
├──── Data Classes              # Type definitions
│    ├──── JobInfo                # Job information
│    ├──── ReturnType             # Message return type
│    └──── JobStatus (Enum)       # Job status enumeration
└──── Exceptions                # Custom exceptions
     └──── ConfigError          # Configuration errors
```

8.3. Development Workflow

8.3.1. Branch Strategy

- `main` - Stable production code
- `develop` - Development integration branch
- `feature/*` - Feature development branches
- `hotfix/*` - Critical bug fixes

8.3.2. Feature Development

```
# Start new feature
git checkout develop
git pull origin develop
git checkout -b feature/my-new-feature

# Make changes and test
# ... development work ...
make test
make lint
```

```
# Commit changes
git add .
git commit -m "feat: add new feature description"

# Push and create PR
git push origin feature/my-new-feature
# Create pull request to develop branch
```

8.3.3. Code Standards

Python Style

- Follow PEP 8 coding standards
- Use type hints for all functions
- Maximum line length: 127 characters
- Use meaningful variable and function names
- Include docstrings for all public methods

Code Quality Checks

```
# Run linting
make lint

# Check specific files
flake8 Listener/event_listener.py

# Security analysis
make security

# Dependency vulnerability check
safety check
```

Testing Requirements

- All new code must have tests
- Maintain 80%+ code coverage
- Include both unit and integration tests
- Test error conditions and edge cases

```
# Run all tests
make test

# Run with coverage
make coverage
```

```
# Run specific test file
pytest tests/test_event_listener.py -v

# Run specific test method
pytest tests/test_event_listener.py::TestJobManagement::test_job_creation -v
```

8.4. Adding New Features

8.4.1. Example: Adding New Configuration Option

1. **Add to EventListenerConfig**:

```
@dataclass(frozen=True)
class EventListenerConfig:
    # ... existing fields ...
    new_option: bool = False  # Add new configuration option
```

2. **Update EventListener to use it**:

```
def __init__(self, config: EventListenerConfig, ...):
    self.config = config
    # Use the new option
    if config.new_option:
        self.enable_new_feature()
```

3. **Add tests**:

```
def test_new_configuration_option():
    """Test new configuration option."""
    config = EventListenerConfig(new_option=True)
    listener = EventListener(config)
    assert listener.config.new_option is True
```

4. **Update documentation**:
   - Add to configuration guide
   - Update API reference
   - Include usage examples

8.4.2. Example: Adding New Job Status

1. **Add to JobStatus enum**:

```
class JobStatus(Enum):
    # ... existing statuses ...
```

```
    NEW_STATUS = "new_status"
```

2. **Handle in job management**:

```
async def handle_new_status(self, job_id: str):
    """Handle new job status."""
    await self._update_job_status(job_id, JobStatus.NEW_STATUS)
```

3. **Add query method**:

```
async def get_new_status_jobs(self) -> Dict[str, JobInfo]:
    """Get jobs with new status."""
    async with self.job_lock:
        return {jid: job for jid, job in self.jobs.items()
                if job.status == JobStatus.NEW_STATUS}
```

8.5. Testing Framework

8.5.1. Test Categories

Category	Description	Marker
Unit Tests	Test individual components	`@pytest.mark.unit`
Integration Tests	Test component interactions	`@pytest.mark.integration`
Slow Tests	Long-running tests	`@pytest.mark.slow`

8.5.2. Writing Tests

Unit Test Example

```
import pytest
from Listener import EventListenerConfig, EventListener

@pytest.mark.unit
class TestEventListenerConfig:
    """Test EventListenerConfig class."""

    def test_default_configuration(self):
        """Test default configuration values."""
        config = EventListenerConfig()
        assert config.host == "localhost"
        assert config.port == 1883
        assert config.auto_reconnect is True

    def test_custom_configuration(self):
```

```
        """Test custom configuration values."""
        config = EventListenerConfig(
            host="mqtt.example.com",
            port=8883,
            auto_reconnect=False
        )
        assert config.host == "mqtt.example.com"
        assert config.port == 8883
        assert config.auto_reconnect is False
```

Integration Test Example

```
import pytest
import asyncio
from unittest.mock import AsyncMock

@pytest.mark.integration
@pytest.mark.asyncio
class TestEventListenerIntegration:
    """Integration tests for EventListener."""

    async def test_message_processing_workflow(self, event_listener):
        """Test complete message processing workflow."""
        # Mock MQTT client
        event_listener.client = AsyncMock()

        # Test data
        test_data = {"job_id": "test-001", "task": "process"}

        # Process message
        result = await event_listener.process_message(test_data)

        # Verify job tracking
        job_info = await event_listener.get_job_status("test-001")
        assert job_info is not None
        assert job_info.status == JobStatus.COMPLETED
```

Async Test Example

```
@pytest.mark.asyncio
async def test_async_job_management(event_listener):
    """Test async job management operations."""
    # Create test job
    await event_listener._create_job("async-001", {"data": "test"})

    # Verify job exists
    exists = await event_listener.job_exists("async-001")
    assert exists is True
```

```
    # Check job status
    is_running = await event_listener.is_job_running("async-001")
    assert is_running is True
```

8.5.3. Test Fixtures

Common test fixtures are defined in `conftest.py`:

```
@pytest.fixture
def sample_config():
    """Sample EventListener configuration."""
    return EventListenerConfig(
        host="test-host",
        port=1883,
        topic="test/topic",
        client_id="test-client"
    )

@pytest.fixture
def event_listener(sample_config):
    """EventListener instance for testing."""
    return EventListener(sample_config)

@pytest.fixture
async def running_listener(event_listener):
    """Running EventListener instance."""
    # Setup
    await event_listener._connect()
    yield event_listener
    # Cleanup
    event_listener.stop()
```

8.6. Build and Release Process

8.6.1. Building the Package

```
# Clean previous builds
rm -rf dist/ build/ *.egg-info/

# Build package
python -m build

# Verify build
ls -la dist/
```

8.6.2. Release Checklist

1. **Pre-release**:
   - ☐ All tests pass
   - ☐ Code coverage meets requirements
   - ☐ Documentation updated
   - ☐ Version numbers updated
   - ☐ CHANGELOG.md updated
2. **Version Update**:
   - ☐ Update `pyproject.toml` version
   - ☐ Update `Listener/init.py` version
   - ☐ Update documentation version references
3. **Create Release**:

```
# Create git tag
git tag -a v1.x.x -m "Release version 1.x.x"
git push origin v1.x.x

# Build and distribute
./scripts/build_package.sh
```

4. **Post-release**:
   - ☐ Verify installation from git tag
   - ☐ Update internal distribution documentation
   - ☐ Notify team of new release

8.7. Debugging and Troubleshooting

8.7.1. Logging Configuration

```
import logging

# Enable debug logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

# Create logger for the library
logger = logging.getLogger('Listener')
logger.setLevel(logging.DEBUG)
```

8.7.2. Common Debug Scenarios

MQTT Connection Issues

```
# Enable MQTT client debugging
import logging
logging.getLogger('amqtt.client').setLevel(logging.DEBUG)

# Test connection manually
config = EventListenerConfig(host="localhost", port=1883)
listener = EventListener(config)
try:
    await listener._connect()
    print("Connection successful")
except Exception as e:
    print(f"Connection failed: {e}")
```

Job Tracking Issues

```
# Monitor job status
async def debug_job_tracking(listener):
    while True:
        all_jobs = await listener.get_all_jobs()
        running = await listener.get_running_jobs()
        completed = await listener.get_completed_jobs()

        print(f"Total jobs: {len(all_jobs)}")
        print(f"Running: {len(running)}")
        print(f"Completed: {len(completed)}")

        await asyncio.sleep(10)
```

8.7.3. Performance Profiling

```
import cProfile
import pstats

def profile_message_processing():
    """Profile message processing performance."""
    pr = cProfile.Profile()
    pr.enable()

    # Run message processing
    asyncio.run(process_test_messages())

    pr.disable()
    stats = pstats.Stats(pr)
```

```
    stats.sort_stats('cumulative').print_stats(20)
```

8.8. Contributing Guidelines

8.8.1. Code Review Process

1. **Pull Request Requirements**:
   - ☐ Descriptive title and description
   - ☐ All tests pass
   - ☐ Code coverage maintained
   - ☐ Documentation updated
   - ☐ No linting errors
2. **Review Checklist**:
   - ☐ Code follows project standards
   - ☐ Tests cover new functionality
   - ☐ Error handling is appropriate
   - ☐ Performance impact considered
   - ☐ Security implications reviewed

8.8.2. Getting Help

- Check existing issues in the repository
- Review test cases for usage examples
- Contact maintainer: aahameed@kth.se
- Review internal documentation

9. Testing Framework

Comprehensive testing guide for the MQTT Event Listener.

9.1. Test Suite Overview

The project maintains high code quality through comprehensive testing:

Metric	Description	Current	Target
Line Coverage	Percentage of code lines tested	83.72%	80%+
Total Tests	All test cases	73	Growing
Unit Tests	Individual component tests	48	Growing
Integration Tests	Component interaction tests	15	Growing

Metric	Description	Current	Target
MQTT Integration	Real broker tests	10	Growing
Performance	Test execution time	~15s	<20s

9.2. Running Tests

9.2.1. Quick Commands

```
# Run all tests with coverage
make test

# Run only unit tests (fast)
pytest -m unit

# Run integration tests with mocks
pytest -m integration

# Run MQTT integration tests (requires broker)
pytest -m mqtt_integration

# Generate coverage report
pytest --cov=Listener --cov-report=html

# Quick development check
pytest -m "unit and not slow"
```

9.2.2. Detailed Test Commands

```
# Run tests with verbose output
pytest -v

# Run specific test file
pytest tests/test_event_listener.py

# Run specific test class
pytest tests/test_event_listener.py::TestJobManagement

# Run specific test method
pytest tests/test_event_listener.py::TestJobManagement::test_job_creation

# Run tests by marker
pytest -m unit                    # Unit tests only
pytest -m integration            # Integration tests only
pytest -m mqtt_integration       # MQTT integration tests
pytest -m "unit and not slow"    # Fast unit tests
```

```
# Run with coverage
pytest --cov=Listener --cov-report=html --cov-report=term-missing

# Run with debugging
pytest -s -vv --tb=long
```

9.3. Test Structure

9.3.1. Test Directory Layout

```
tests/
├──── __init__.py                # Test package initialization
├──── conftest.py                # Shared fixtures and configuration
├──── test_event_listener.py     # EventListener class tests (48 tests)
├──── test_safe_config_parser.py  # SafeConfigParser tests (15 tests)
├──── test_mqtt_integration.py    # MQTT integration tests (10 tests)
└──── README.md                  # Testing documentation
```

9.3.2. Test Categories

Unit Tests (`@pytest.mark.unit`)

Test individual components in isolation with comprehensive coverage:

```
@pytest.mark.unit
class TestEventListenerConfig:
    """Unit tests for EventListenerConfig dataclass."""

    def test_default_values(self):
        """Test default configuration values."""
        config = EventListenerConfig()
        assert config.host == "localhost"
        assert config.port == 1883
        assert config.auto_reconnect is True
        assert config.max_jobs_in_memory == 1000

    def test_custom_values(self):
        """Test custom configuration values."""
        config = EventListenerConfig(
            host="custom.host",
            port=8883,
            max_jobs_in_memory=500
        )
        assert config.host == "custom.host"
        assert config.port == 8883
        assert config.max_jobs_in_memory == 500
```

```
@pytest.mark.unit
class TestJobManagement:
    """Comprehensive job management testing."""

    @pytest.mark.asyncio
    async def test_job_lifecycle(self, event_listener):
        """Test complete job lifecycle."""
        # Create job
        await event_listener._create_job("test-001", {"data": "test"})

        # Verify pending status
        job = await event_listener.get_job_status("test-001")
        assert job.status == JobStatus.PENDING

        # Update to running
        await event_listener._update_job_status("test-001", JobStatus.RUNNING)
        job = await event_listener.get_job_status("test-001")
        assert job.status == JobStatus.RUNNING

        # Complete job
        result = {"success": True, "processed": "test"}
        await event_listener._update_job_status("test-001", JobStatus.COMPLETED,
result=result)
        job = await event_listener.get_job_status("test-001")
        assert job.status == JobStatus.COMPLETED
        assert job.result == result
```

Integration Tests (`@pytest.mark.integration`)

Test component interactions with mocked dependencies:

```
@pytest.mark.integration
@pytest.mark.asyncio
class TestEventListenerIntegration:
    """Integration tests with mocked MQTT client."""

    async def test_complete_workflow(self, event_listener):
        """Test complete message processing workflow."""
        # Mock MQTT client
        event_listener.client = AsyncMock()

        # Test processor function
        def test_processor(data, job_id):
            return ReturnType(
                data={"result": "processed", "input": data},
                topic="test/results",
                qos=0,
                retain=False,
                message_id=1,
                timestamp=datetime.now(),
```

```
                job_id=job_id
            )

        # Process test message
        test_data = {"job_id": "integration-001", "task": "test"}
        await event_listener._process_message(test_data, test_processor)

        # Verify job completion
        job = await event_listener.get_job_status("integration-001")
        assert job.status == JobStatus.COMPLETED

        # Verify result publishing
        event_listener.client.publish.assert_called_once()
        publish_call = event_listener.client.publish.call_args
        assert publish_call[0][0] == "test/results"  # topic
```

MQTT Integration Tests (`@pytest.mark.mqtt_integration`)

NEW: Comprehensive tests against real Mosquitto broker:

```
@pytest.mark.mqtt_integration
@pytest.mark.asyncio
class TestEventListenerMQTTIntegration:
    """Real MQTT broker integration tests."""

    async def test_eventlistener_connection(self, mqtt_username, mqtt_password):
        """Test EventListener connection to real broker."""
        config = EventListenerConfig(
            host="mosquitto",
            port=1883,
            username=mqtt_username,
            password=mqtt_password,
            topic="test/events"
        )

        listener = EventListener(config)
        await listener._connect()
        assert listener.client is not None
        await listener._disconnect()

    async def test_eventlistener_subscription(self, authenticated_listener):
        """Test EventListener subscription functionality."""
        await authenticated_listener._subscribe()
        # Verify subscription was successful
        assert authenticated_listener.client is not None

    async def test_eventlistener_publish_and_job_tracking(self,
authenticated_listener):
        """Test publishing with job tracking."""
        # Create and track a job
```

```
        await authenticated_listener._create_job("mqtt-test-001", {"task": "test"})

        # Publish result
        result_data = {"job_id": "mqtt-test-001", "result": "success"}
        await authenticated_listener._publish_result(result_data, "test/results")

        # Verify job status
        job = await authenticated_listener.get_job_status("mqtt-test-001")
        assert job is not None

    async def test_toml_message_processing(self, authenticated_listener):
        """Test processing of TOML-formatted messages."""
        toml_message = '''
        job_id = "toml-test-001"
        task_type = "data_processing"

        [data]
        input_file = "/test/input.csv"
        output_file = "/test/output.json"
        '''

        # Process TOML message
        config_parser = SafeConfigParser()
        parsed_data = config_parser.parse_config_from_string(toml_message)

        await authenticated_listener._create_job(parsed_data["job_id"], parsed_data)

        # Verify job creation
        job = await authenticated_listener.get_job_status("toml-test-001")
        assert job is not None
        assert job.job_id == "toml-test-001"
```

Slow Tests (`@pytest.mark.slow`)

Long-running tests for comprehensive scenarios:

```
@pytest.mark.slow
@pytest.mark.asyncio
async def test_memory_management_over_time(event_listener):
    """Test memory management with many jobs over time."""
    # Configure for fast cleanup
    config = EventListenerConfig(
        max_jobs_in_memory=100,
        job_cleanup_interval=1  # 1 second cleanup
    )
    listener = EventListener(config)

    # Create many jobs
    for i in range(200):
        await listener._create_job(f"job-{i}", {"data": i})
```

```
    # Wait for cleanup
    await asyncio.sleep(2)

    # Verify memory management
    all_jobs = await listener.get_all_jobs()
    assert len(all_jobs) <= 100
```

9.4. Test Fixtures

9.4.1. MQTT Authentication Fixtures

NEW: Fixtures for authenticated MQTT testing:

```
@pytest.fixture
def mqtt_username():
    """MQTT username from environment."""
    return os.getenv("MQTT_USERNAME", "user")

@pytest.fixture
def mqtt_password():
    """MQTT password from environment."""
    return os.getenv("MQTT_PASSWORD", "password")

@pytest.fixture
async def authenticated_listener(mqtt_username, mqtt_password):
    """EventListener connected to real MQTT broker."""
    config = EventListenerConfig(
        host="mosquitto",
        port=1883,
        username=mqtt_username,
        password=mqtt_password,
        topic="test/events"
    )

    listener = EventListener(config)
    await listener._connect()
    yield listener
    await listener._disconnect()
```

9.4.2. Configuration Fixtures

```
@pytest.fixture
def sample_config():
    """Sample configuration for testing."""
    return EventListenerConfig(
        host="test-broker",
```

```
        port=1883,
        topic="test/events",
        client_id="test-client",
        max_jobs_in_memory=100,
        job_cleanup_interval=60
    )

@pytest.fixture
def ssl_config():
    """SSL configuration for testing."""
    return EventListenerConfig(
        host="ssl-broker",
        port=8883,
        cafile="/path/to/test-ca.crt",
        topic="secure/events"
    )
```

9.4.3. EventListener Fixtures

```
@pytest.fixture
def event_listener(sample_config):
    """EventListener instance for testing."""
    return EventListener(sample_config)

@pytest.fixture
def mock_mqtt_client():
    """Mocked MQTT client."""
    client = AsyncMock()
    client.connect = AsyncMock()
    client.disconnect = AsyncMock()
    client.subscribe = AsyncMock()
    client.publish = AsyncMock()
    return client

@pytest.fixture
async def connected_listener(event_listener, mock_mqtt_client):
    """EventListener with mocked MQTT client."""
    event_listener.client = mock_mqtt_client
    await event_listener._connect()
    yield event_listener
    event_listener.stop()
```

9.4.4. Data Fixtures

```
@pytest.fixture
def sample_toml_data():
    """Sample TOML data for testing."""
    return {
```

```
        "job_id": "test-001",
        "task_type": "data_processing",
        "priority": "high",
        "data": {
            "input_file": "/test/input.csv",
            "output_file": "/test/output.json"
        }
    }

@pytest.fixture
def temp_toml_file(tmp_path):
    """Temporary TOML file for testing."""
    toml_content = """
    job_id = "file-test-001"
    task_type = "file_processing"

    [data]
    input = "/path/to/input"
    output = "/path/to/output"
    """

    toml_file = tmp_path / "test_config.toml"
    toml_file.write_text(toml_content)
    return str(toml_file)
```

9.5. Testing Patterns

9.5.1. Async Testing

```
@pytest.mark.asyncio
async def test_async_job_operations(event_listener):
    """Test async job management operations."""
    # Create job
    await event_listener._create_job("async-001", {"test": "data"})

    # Check job status
    job_info = await event_listener.get_job_status("async-001")
    assert job_info is not None
    assert job_info.status == JobStatus.PENDING

    # Update job status
    await event_listener._update_job_status(
        "async-001",
        JobStatus.COMPLETED,
        result={"success": True}
    )

    # Verify update
    updated_job = await event_listener.get_job_status("async-001")
```

```
    assert updated_job.status == JobStatus.COMPLETED
    assert updated_job.result == {"success": True}
```

9.5.2. Mocking External Dependencies

```
from unittest.mock import AsyncMock, patch

@pytest.mark.unit
async def test_mqtt_connection_error_handling(event_listener):
    """Test MQTT connection error handling."""
    # Mock connection failure
    with patch.object(event_listener.client, 'connect') as mock_connect:
        mock_connect.side_effect = ConnectionError("Connection failed")

        # Test error handling
        with pytest.raises(ConnectionError):
            await event_listener._connect()

@pytest.mark.unit
def test_toml_parsing_error_handling(safe_config_parser):
    """Test TOML parsing error handling."""
    invalid_toml = "invalid toml content [[[["

    with pytest.raises(ConfigError):
        safe_config_parser.parse_config_from_string(invalid_toml)
```

9.5.3. Parameterized Tests

```
@pytest.mark.parametrize("host,port,expected", [
    ("localhost", 1883, "mqtt://localhost:1883"),
    ("mqtt.example.com", 8883, "mqtt://mqtt.example.com:8883"),
    ("ssl-broker", 8884, "mqtt://ssl-broker:8884"),
])
def test_uri_generation(host, port, expected):
    """Test URI generation with different parameters."""
    config = EventListenerConfig(host=host, port=port)
    assert config.uri == expected

@pytest.mark.parametrize("status,expected_running", [
    (JobStatus.PENDING, False),
    (JobStatus.RUNNING, True),
    (JobStatus.COMPLETED, False),
    (JobStatus.FAILED, False),
])
async def test_job_running_status(event_listener, status, expected_running):
    """Test job running status detection."""
    await event_listener._create_job("param-001", {})
    await event_listener._update_job_status("param-001", status)
```

```
    is_running = await event_listener.is_job_running("param-001")
    assert is_running == expected_running
```

9.5.4. Error Condition Testing

```
@pytest.mark.unit
def test_invalid_configuration():
    """Test handling of invalid configuration."""
    with pytest.raises(ValueError):
        EventListenerConfig(port=-1)  # Invalid port

    with pytest.raises(ValueError):
        EventListenerConfig(qos=5)    # Invalid QoS

@pytest.mark.unit
async def test_duplicate_job_handling(event_listener):
    """Test duplicate job detection and handling."""
    # Create initial job
    await event_listener._create_job("dup-001", {"data": "test"})

    # Try to create duplicate
    duplicate_created = await event_listener._create_job("dup-001", {"data": "test2"})
    assert duplicate_created is False

    # Verify duplicate tracking
    duplicates = await event_listener.get_duplicate_jobs()
    assert "dup-001" in duplicates
```

9.6. Coverage Analysis

9.6.1. Coverage Configuration

Coverage settings in `pyproject.toml`:

```
[tool.coverage.run]
source = ["Listener"]
omit = ["*/tests/*", "*/test_*"]

[tool.coverage.report]
exclude_lines = [
    "pragma: no cover",
    "def __repr__",
    "if self.debug:",
    "raise AssertionError",
    "raise NotImplementedError",
    "if __name__ == .__main__.:",
```

```
]
```

9.6.2. Coverage Reports

```
# Generate HTML coverage report
pytest --cov=Listener --cov-report=html
open htmlcov/index.html

# Generate terminal report
pytest --cov=Listener --cov-report=term-missing

# Generate XML report (for CI)
pytest --cov=Listener --cov-report=xml
```

9.6.3. Coverage Targets

Coverage Type	Description	Current	Target
Line Coverage	Percentage of lines executed	83.72%	80%+
Branch Coverage	Percentage of branches taken	82%	80%+
Function Coverage	Percentage of functions called	95%	90%+

9.7. Performance Testing

9.7.1. Benchmark Tests

```
import time
import pytest

@pytest.mark.benchmark
def test_job_creation_performance(event_listener, benchmark):
    """Benchmark job creation performance."""
    def create_jobs():
        for i in range(100):
            asyncio.run(event_listener._create_job(f"perf-{i}", {"data": i}))

    result = benchmark(create_jobs)
    assert result is not None

@pytest.mark.benchmark
async def test_message_processing_throughput(event_listener):
    """Test message processing throughput."""
    start_time = time.time()

    # Process 1000 messages
    for i in range(1000):
```

```
        test_data = {"job_id": f"throughput-{i}", "data": i}
        await event_listener._process_message(test_data)

    end_time = time.time()
    duration = end_time - start_time
    throughput = 1000 / duration

    print(f"Throughput: {throughput:.1f} messages/second")
    assert throughput > 100  # Minimum acceptable throughput
```

9.7.2. Memory Usage Tests

```
import psutil
import gc

@pytest.mark.slow
async def test_memory_usage_under_load(event_listener):
    """Test memory usage under sustained load."""
    process = psutil.Process()
    initial_memory = process.memory_info().rss

    # Create many jobs
    for i in range(10000):
        await event_listener._create_job(f"memory-{i}", {"data": f"test-{i}"})

    # Force garbage collection
    gc.collect()

    peak_memory = process.memory_info().rss
    memory_increase = peak_memory - initial_memory
    memory_mb = memory_increase / 1024 / 1024

    print(f"Memory increase: {memory_mb:.1f} MB")
    assert memory_mb < 100  # Should not use more than 100MB
```

9.8. Continuous Integration

9.8.1. GitHub Actions Workflow

Tests run automatically on:

- Push to `main` or `develop` branches
- Pull requests
- Tag creation

```
name: Tests
```

```yaml
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.8, 3.9, 3.10, 3.11, 3.12]

    steps:
    - uses: actions/checkout@v3
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install dependencies
      run: |
        pip install -e .[dev]

    - name: Run tests
      run: |
        pytest --cov=Listener --cov-report=xml

    - name: Upload coverage
      uses: codecov/codecov-action@v3
```

9.9. Test Maintenance

9.9.1. Adding New Tests

When adding new functionality:

1. **Write tests first** (TDD approach)
2. **Cover both success and failure cases**
3. **Include edge cases and error conditions**
4. **Add integration tests for new workflows**
5. **Update test documentation**

9.9.2. Test Review Checklist

1. [ ] Tests cover new functionality
2. [ ] Both positive and negative cases tested
3. [ ] Async operations properly tested
4. [ ] Mocks used appropriately

5. [ ] Test names are descriptive
6. [ ] Tests are fast and reliable
7. [ ] Coverage requirements met

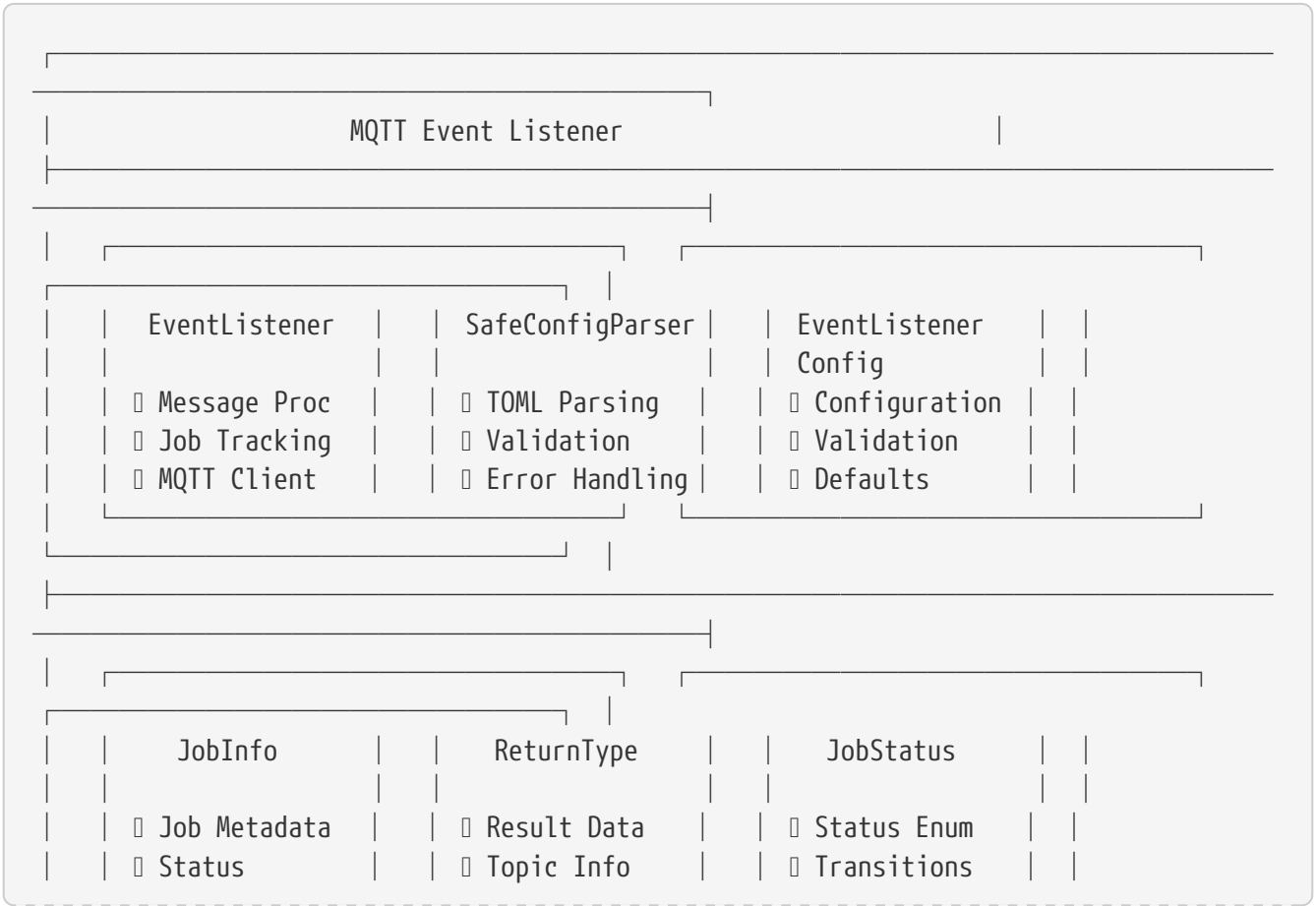
9.9.3. Debugging Tests

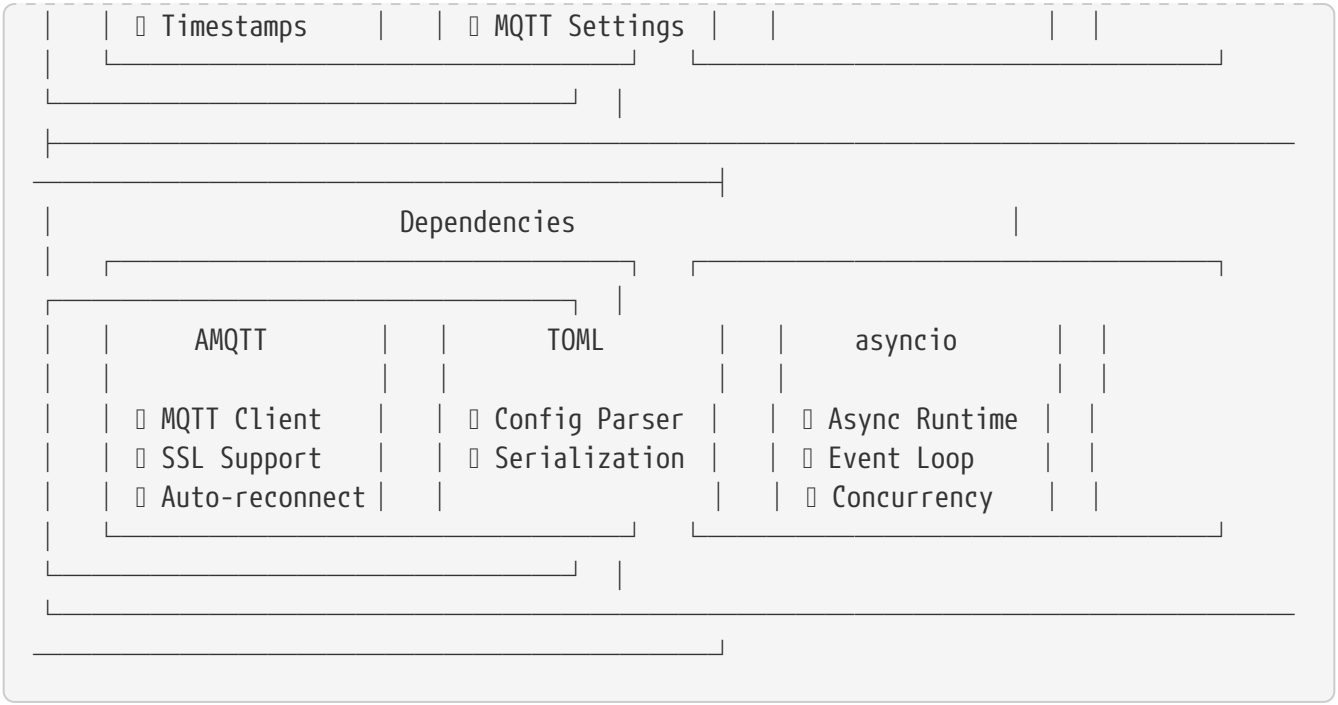
```
# Run single test with debugging
pytest tests/test_event_listener.py::test_specific -s -vv

# Run tests and drop into debugger on failure
pytest --pdb

# Run tests with profiling
pytest --profile

# Show test durations
pytest --durations=10
```

10. System Architecture

Technical architecture and design of the MQTT Event Listener.

10.1. High-Level Architecture





10.2. Component Architecture

10.2.1. EventListener Core

The main `EventListener` class orchestrates all functionality:

```
EventListener
 ├─── Configuration Management
 │    ├─── Config validation
 │    ├─── Environment integration
 │    └─── Default value handling
 ├─── MQTT Client Management
 │    ├─── Connection handling
 │    ├─── Subscription management
 │    ├─── Auto-reconnection
 │    └─── SSL/TLS support
 ├─── Message Processing
 │    ├─── TOML parsing
 │    ├─── Job creation
 │    ├─── Function execution
 │    └─── Result publishing
 ├─── Job Tracking System
 │    ├─── In-memory storage
 │    ├─── Status management
 │    ├─── Duplicate detection
 │    └─── Cleanup operations
 └─── Error Handling
      ├─── Connection errors
      ├─── Processing errors
      ├─── Configuration errors
```

```
    └──── Recovery mechanisms
```

10.2.2. Data Flow

```
MQTT Message → TOML Parser → Job Creation → Processing Function → Result Publishing
     │              │              │              │                    │
Connection     Validation    Job Tracking   User Function     MQTT Publish
Management     & Parsing     & Status       Execution         & Logging
     │              │              │              │                    │
Auto-reconnect Error         Memory         Error            Topic
& Monitoring   Handling      Management     Handling         Routing
```

10.3. Memory Management

10.3.1. Job Storage Strategy

Component	Storage Type	Lifetime	Cleanup Strategy
Active Jobs	In-memory Dict	Processing duration	Immediate on completion
Completed Jobs	In-memory Dict	Configurable TTL	Periodic cleanup
Failed Jobs	In-memory Dict	Configurable TTL	Periodic cleanup
Duplicate Jobs	In-memory Set	Configurable TTL	Periodic cleanup

10.3.2. Memory Optimization

- **Configurable limits** on maximum jobs in memory
- **Periodic cleanup** of old completed jobs
- **Efficient data structures** for job tracking
- **Memory monitoring** capabilities

```
# Memory-efficient job storage
class JobTracker:
    def __init__(self, max_jobs: int = 5000):
        self.jobs: Dict[str, JobInfo] = {}
        self.max_jobs = max_jobs
        self.cleanup_interval = 3600  # 1 hour

    async def cleanup_old_jobs(self):
        """Remove old completed/failed jobs."""
        cutoff_time = datetime.now() - timedelta(seconds=self.cleanup_interval)
```

```
        to_remove = [
            job_id for job_id, job in self.jobs.items()
            if job.status in [JobStatus.COMPLETED, JobStatus.FAILED]
            and job.completed_at
            and job.completed_at < cutoff_time
        ]

        for job_id in to_remove:
            del self.jobs[job_id]
```

10.4. Concurrency Model

10.4.1. Async/Await Architecture

The library is built on Python's asyncio framework:

```
Main Event Loop
 ├──── MQTT Client Task
 │    ├──── Connection monitoring
 │    ├──── Message receiving
 │    └──── Reconnection handling
 ├──── Message Processing Tasks
 │    ├──── Parallel processing
 │    ├──── Job status updates
 │    └──── Result publishing
 ├──── Cleanup Tasks
 │    ├──── Periodic job cleanup
 │    ├──── Memory monitoring
 │    └──── Health checks
 └──── User Function Execution
     ├──── Sync function wrapper
     ├──── Async function direct call
     └──── Error boundary handling
```

10.4.2. Thread Safety

- **asyncio-based** - Single-threaded event loop
- **Lock-protected** job management operations
- **Atomic updates** for job status changes
- **Thread-safe** configuration access

```
class EventListener:
    def __init__(self, config: EventListenerConfig):
        self.job_lock = asyncio.Lock()  # Protects job operations
        self.jobs: Dict[str, JobInfo] = {}
```

```
    async def _create_job(self, job_id: str, data: Dict[str, Any]) -> bool:
        async with self.job_lock:
            if job_id in self.jobs:
                return False  # Duplicate
            self.jobs[job_id] = JobInfo(job_id=job_id, ...)
            return True
```

10.5. Error Handling Strategy

10.5.1. Error Classification

Error Type	Description	Recovery	Impact
Connection Errors	MQTT broker connectivity	Auto-reconnect	Service disruption
Configuration Errors	Invalid settings	Manual fix required	Startup failure
Processing Errors	User function failures	Job marked failed	Single job impact
Memory Errors	Resource exhaustion	Cleanup + backpressure	Performance degradation

10.5.2. Error Recovery

```
Error Detection
     
Error Classification
     
Recovery Strategy Selection
      ├──── Auto-retry (Connection)
      ├──── Job Failure (Processing)
      ├──── Circuit Breaker (Repeated failures)
      └──── Graceful Degradation (Resource issues)
     
Error Logging & Monitoring
     
Recovery Execution
```

10.6. Performance Characteristics

10.6.1. Throughput Metrics

Metric	Description	Typical Value	Target
Message Processing	Messages per second	100-1000 msg/s	>100 msg/s
Memory Usage	RAM per 1000 jobs	~10-50 MB	<100 MB
Latency	Processing delay	<100ms	<500ms
Connection Recovery	Reconnection time	1-10 seconds	<30 seconds

10.6.2. Scalability Factors

- **Message size** - Larger TOML messages require more parsing time
- **Processing complexity** - User function execution time dominates
- **Job retention** - More jobs in memory increase lookup time
- **Network latency** - MQTT broker distance affects performance

10.7. Security Architecture

10.7.1. Security Layers

```
Application Security
 ├─── Input Validation
 │    ├─── TOML syntax validation
 │    ├─── Configuration validation
 │    └─── Message size limits
 ├─── Connection Security
 │    ├─── SSL/TLS encryption
 │    ├─── Certificate validation
 │    ├─── Authentication
 │    └─── Authorization
 ├─── Data Security
 │    ├─── Secure configuration handling
 │    ├─── Credential management
 │    └─── Memory protection
 └─── Process Security
      ├─── Error information filtering
      ├─── Logging sanitization
      └─── Resource limits
```

10.7.2. Security Considerations

- **Credential Protection** - Environment variables for sensitive data
- **Network Security** - SSL/TLS for all MQTT connections
- **Input Validation** - Strict TOML parsing and validation

- **Resource Limits** - Memory and processing bounds
- **Error Handling** - No sensitive data in error messages

10.8. Configuration Architecture

10.8.1. Configuration Hierarchy

```
Configuration Sources (Priority Order)
1. Explicit Parameters (Highest)
2. Environment Variables
3. Configuration Files
4. Default Values (Lowest)

Configuration Validation
 ├──── Type checking
 ├──── Range validation
 ├──── Format validation
 └──── Dependency validation

Configuration Application
 ├──── MQTT client setup
 ├──── Job management setup
 ├──── Error handling setup
 └──── Performance tuning
```

10.8.2. Configuration Flow

```
# Configuration processing flow
EventListenerConfig.__post_init__()
    
validate_configuration()
     ├──── Port range (1-65535)
     ├──── QoS values (0, 1, 2)
     ├──── Timeout values (>0)
     └──── Memory limits (>0)
    
apply_environment_overrides()
    
generate_derived_values()
     ├──── URI from host/port
     ├──── Topic configurations
     └──── SSL context setup
```

10.9. Extensibility Points

10.9.1. Plugin Architecture

The library provides several extension points:

- **Custom Processors** - User-defined message processing functions
- **Configuration Parsers** - Alternative to SafeConfigParser
- **Error Handlers** - Custom error processing logic
- **Monitoring Hooks** - Performance and health monitoring

10.9.2. Future Extensions

- **Persistence Layer** - Database storage for jobs
- **Message Serializers** - Support for JSON, MessagePack, etc.
- **Load Balancing** - Distributed processing capabilities
- **Caching Layer** - Redis/Memcached integration
- **Metrics Export** - Prometheus/StatsD integration

10.10. Testing Architecture

10.10.1. Test Strategy

```
Testing Pyramid
 ├──── Unit Tests (45+ tests)
 │    ├──── Configuration validation
 │    ├──── Job management
 │    ├──── TOML parsing
 │    └──── Error handling
 ├──── Integration Tests (12+ tests)
 │    ├──── MQTT connectivity
 │    ├──── End-to-end workflows
 │    ├──── Error scenarios
 │    └──── Performance tests
 └──── System Tests
     ├──── Multi-environment
     ├──── Load testing
     └──── Failure testing
```

10.10.2. Mock Architecture

- **MQTT Client Mocking** - AsyncMock for AMQTT client
- **Network Simulation** - Connection failure scenarios
- **Time Mocking** - Accelerated cleanup testing
- **File System Mocking** - Configuration file testing

10.11. Deployment Architecture

10.11.1. Internal Deployment

```
Development Environment
 ├──── Local MQTT broker
 ├──── Development configuration
 ├──── Debug logging
 └──── Test data

Staging Environment
 ├──── Staging MQTT broker
 ├──── Production-like config
 ├──── Integration testing
 └──── Performance validation

Production Environment
 ├──── Production MQTT broker
 ├──── Optimized configuration
 ├──── Monitoring & alerting
 └──── High availability
```

10.11.2. Distribution Model

- **Git-based Installation** - Direct from repository
- **Wheel Distribution** - Pre-built packages
- **Container Support** - Docker deployment ready
- **Internal Package Index** - Private package repository

10.12. Monitoring and Observability

10.12.1. Built-in Monitoring

- **Job Status Tracking** - Real-time job state monitoring
- **Connection Health** - MQTT connection status
- **Memory Usage** - Job memory consumption
- **Error Rates** - Processing failure statistics

10.12.2. External Integration

- **Logging Integration** - Standard Python logging
- **Metrics Export** - Future Prometheus support
- **Health Checks** - HTTP endpoint (planned)

- **Distributed Tracing** - OpenTelemetry (planned)

11. Examples and Use Cases

Practical examples for common use cases of the MQTT Event Listener.

11.1. Basic Examples

11.1.1. Simple Message Processing

```
import asyncio
from Listener import EventListener, EventListenerConfig

async def main():
    """Basic message processing example."""
    # Configuration
    config = EventListenerConfig(
        host="localhost",
        port=1883,
        topic="events/tasks",
        client_id="simple-processor"
    )

    # Create listener
    listener = EventListener(config)

    # Define processor
    def simple_processor(data, job_id):
        """Process incoming messages."""
        print(f"Processing job {job_id}")
        print(f"Task: {data.get('task_type', 'unknown')}")

        # Simple processing logic
        result = {
            "job_id": job_id,
            "status": "completed",
            "processed_at": datetime.now().isoformat(),
            "data": data
        }

        return ReturnType(
            data=result,
            topic="events/results",
            qos=1,
            retain=False,
            message_id=int(time.time()),
            timestamp=datetime.now(),
            job_id=job_id
```

```
        )

    # Start processing
    print("Starting MQTT Event Listener...")
    await listener.run(simple_processor)

if __name__ == "__main__":
    asyncio.run(main())
```

11.1.2. Configuration from Environment

```
import os
import asyncio
from Listener import EventListener, EventListenerConfig

def create_config_from_env():
    """Create configuration from environment variables."""
    return EventListenerConfig(
        host=os.getenv("MQTT_HOST", "localhost"),
        port=int(os.getenv("MQTT_PORT", 1883)),
        username=os.getenv("MQTT_USERNAME"),
        password=os.getenv("MQTT_PASSWORD"),
        topic=os.getenv("MQTT_TOPIC", "events"),
        client_id=os.getenv("MQTT_CLIENT_ID", "env-listener"),

        # SSL settings
        cafile=os.getenv("MQTT_CA_FILE"),

        # Job settings
        max_jobs_in_memory=int(os.getenv("MAX_JOBS", 5000)),
        job_cleanup_interval=int(os.getenv("CLEANUP_INTERVAL", 3600))
    )

async def main():
    config = create_config_from_env()
    listener = EventListener(config)

    def env_processor(data, job_id):
        return ReturnType(
            data={"processed": True, "job_id": job_id},
            topic=os.getenv("RESULTS_TOPIC", "results"),
            qos=1,
            retain=False,
            message_id=1,
            timestamp=datetime.now(),
            job_id=job_id
        )

    await listener.run(env_processor)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

11.2. Advanced Examples

11.2.1. Multi-Topic Processing

```
import asyncio
from Listener import EventListener, EventListenerConfig, ReturnType

async def multi_topic_example():
    """Process messages from multiple topics with different handling."""

    config = EventListenerConfig(
        host="mqtt.example.com",
        port=1883,
        topic="events/#",  # Subscribe to all events
        custom_topics={
            "events/alerts": {"qos": 2, "retain": True},
            "events/metrics": {"qos": 0, "retain": False},
            "events/commands": {"qos": 1, "retain": False}
        }
    )

    listener = EventListener(config)

    def multi_topic_processor(data, job_id):
        """Route processing based on topic or message type."""
        topic = data.get('_topic', '')  # Topic info from context
        message_type = data.get('type', 'unknown')

        if 'alerts' in topic or message_type == 'alert':
            return process_alert(data, job_id)
        elif 'metrics' in topic or message_type == 'metric':
            return process_metric(data, job_id)
        elif 'commands' in topic or message_type == 'command':
            return process_command(data, job_id)
        else:
            return process_default(data, job_id)

    def process_alert(data, job_id):
        """Process alert messages with high priority."""
        severity = data.get('severity', 'info')
        message = data.get('message', 'No message')

        print(f" ALERT [{severity.upper()}]: {message}")

        # Send to monitoring system
```

```
        result = {
            "type": "alert_processed",
            "job_id": job_id,
            "severity": severity,
            "processed_at": datetime.now().isoformat(),
            "alert_id": data.get('alert_id')
        }

        return ReturnType(
            data=result,
            topic="monitoring/alerts",
            qos=2,  # High reliability for alerts
            retain=True,
            message_id=int(time.time()),
            timestamp=datetime.now(),
            job_id=job_id
        )

    def process_metric(data, job_id):
        """Process metric data for analytics."""
        metric_name = data.get('metric_name')
        value = data.get('value')
        timestamp = data.get('timestamp')

        print(f"□ METRIC: {metric_name} = {value} at {timestamp}")

        result = {
            "type": "metric_processed",
            "job_id": job_id,
            "metric": metric_name,
            "value": value,
            "processed_at": datetime.now().isoformat()
        }

        return ReturnType(
            data=result,
            topic="analytics/metrics",
            qos=0,  # Lower reliability for metrics
            retain=False,
            message_id=int(time.time()),
            timestamp=datetime.now(),
            job_id=job_id
        )

    def process_command(data, job_id):
        """Process command messages."""
        command = data.get('command')
        params = data.get('parameters', {})

        print(f"□ COMMAND: {command} with params {params}")
```

```
        # Execute command (simplified)
        success = execute_command(command, params)

        result = {
            "type": "command_executed",
            "job_id": job_id,
            "command": command,
            "success": success,
            "executed_at": datetime.now().isoformat()
        }

        return ReturnType(
            data=result,
            topic="commands/results",
            qos=1,
            retain=False,
            message_id=int(time.time()),
            timestamp=datetime.now(),
            job_id=job_id
        )

    def process_default(data, job_id):
        """Default processing for unknown message types."""
        print(f"□ DEFAULT: Processing job {job_id}")

        result = {
            "type": "default_processed",
            "job_id": job_id,
            "original_data": data,
            "processed_at": datetime.now().isoformat()
        }

        return ReturnType(
            data=result,
            topic="events/processed",
            qos=0,
            retain=False,
            message_id=int(time.time()),
            timestamp=datetime.now(),
            job_id=job_id
        )

    def execute_command(command, params):
        """Simulate command execution."""
        # Simplified command execution
        commands = {
            "restart": lambda: True,
            "status": lambda: True,
            "backup": lambda: params.get('target') is not None
        }
```

```
        handler = commands.get(command)
        return handler() if handler else False

    # Start processing
    await listener.run(multi_topic_processor)

if __name__ == "__main__":
    asyncio.run(multi_topic_example())
```

11.2.2. Error Handling and Resilience

```
import asyncio
import logging
from datetime import datetime, timedelta
from Listener import EventListener, EventListenerConfig, ReturnType, ConfigError

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

async def resilient_processor_example():
    """Example with comprehensive error handling and resilience."""

    config = EventListenerConfig(
        host="mqtt.example.com",
        port=1883,
        topic="tasks/+",
        auto_reconnect=True,
        reconnect_retries=10,
        reconnect_max_interval=60,
        max_jobs_in_memory=1000,
        job_cleanup_interval=1800  # 30 minutes
    )

    class ResilientProcessor:
        def __init__(self):
            self.error_count = 0
            self.last_error_time = None
            self.circuit_breaker_open = False
            self.max_errors = 5
            self.circuit_reset_time = timedelta(minutes=5)

        def check_circuit_breaker(self):
            """Simple circuit breaker implementation."""
            if self.circuit_breaker_open:
                if (datetime.now() - self.last_error_time) > self.circuit_reset_time:
                    self.circuit_breaker_open = False
                    self.error_count = 0
                    logger.info("Circuit breaker reset")
```

```
                return False
            return True
        return False

    def record_error(self):
        """Record error for circuit breaker."""
        self.error_count += 1
        self.last_error_time = datetime.now()

        if self.error_count >= self.max_errors:
            self.circuit_breaker_open = True
            logger.warning("Circuit breaker opened due to errors")

    def record_success(self):
        """Record successful processing."""
        if self.error_count > 0:
            self.error_count = max(0, self.error_count - 1)

    def process_message(self, data, job_id):
        """Process message with error handling."""
        try:
            # Check circuit breaker
            if self.check_circuit_breaker():
                return self.create_error_response(
                    job_id,
                    "circuit_breaker",
                    "Circuit breaker is open"
                )

            # Validate input
            if not self.validate_input(data):
                return self.create_error_response(
                    job_id,
                    "validation_error",
                    "Invalid input data"
                )

            # Process based on task type
            task_type = data.get('task_type', 'unknown')

            if task_type == 'data_processing':
                result = self.process_data_task(data, job_id)
            elif task_type == 'file_operation':
                result = self.process_file_task(data, job_id)
            elif task_type == 'api_call':
                result = self.process_api_task(data, job_id)
            else:
                result = self.process_unknown_task(data, job_id)

            # Record success
            self.record_success()
```

```
                return result

            except ValueError as e:
                logger.error(f"Validation error for job {job_id}: {e}")
                return self.create_error_response(job_id, "validation_error", str(e))

            except ConnectionError as e:
                logger.error(f"Connection error for job {job_id}: {e}")
                self.record_error()
                return self.create_error_response(job_id, "connection_error", str(e))

            except TimeoutError as e:
                logger.error(f"Timeout for job {job_id}: {e}")
                return self.create_error_response(job_id, "timeout_error", str(e))

            except Exception as e:
                logger.exception(f"Unexpected error for job {job_id}")
                self.record_error()
                return self.create_error_response(job_id, "processing_error", str(e))

        def validate_input(self, data):
            """Validate input data."""
            required_fields = ['task_type', 'job_id']
            return all(field in data for field in required_fields)

        def process_data_task(self, data, job_id):
            """Process data processing task."""
            input_data = data.get('input_data', {})
            operation = data.get('operation', 'transform')

            # Simulate processing
            if operation == 'transform':
                result_data = {key: str(value).upper() for key, value in
input_data.items()}
            elif operation == 'aggregate':
                result_data = {"count": len(input_data), "keys":
list(input_data.keys())}
            else:
                raise ValueError(f"Unknown operation: {operation}")

            return self.create_success_response(job_id, result_data, "data/results")

        def process_file_task(self, data, job_id):
            """Process file operation task."""
            file_path = data.get('file_path')
            operation = data.get('operation', 'read')

            if not file_path:
                raise ValueError("file_path is required for file operations")

            # Simulate file operation
```

```
            if operation == 'read':
                result = {"operation": "read", "file": file_path, "size": 1024}
            elif operation == 'write':
                result = {"operation": "write", "file": file_path, "success": True}
            else:
                raise ValueError(f"Unknown file operation: {operation}")

            return self.create_success_response(job_id, result, "files/results")

        def process_api_task(self, data, job_id):
            """Process API call task."""
            endpoint = data.get('endpoint')
            method = data.get('method', 'GET')

            if not endpoint:
                raise ValueError("endpoint is required for API calls")

            # Simulate API call (could raise ConnectionError, TimeoutError)
            result = {
                "endpoint": endpoint,
                "method": method,
                "status_code": 200,
                "response": {"success": True}
            }

            return self.create_success_response(job_id, result, "api/results")

        def process_unknown_task(self, data, job_id):
            """Process unknown task type."""
            logger.warning(f"Unknown task type for job {job_id}:
{data.get('task_type')}")

            result = {
                "job_id": job_id,
                "status": "skipped",
                "reason": "unknown_task_type",
                "original_data": data
            }

            return ReturnType(
                data=result,
                topic="tasks/skipped",
                qos=0,
                retain=False,
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

        def create_success_response(self, job_id, result_data, topic):
            """Create success response."""
```

```
            result = {
                "job_id": job_id,
                "status": "completed",
                "result": result_data,
                "processed_at": datetime.now().isoformat()
            }

            return ReturnType(
                data=result,
                topic=topic,
                qos=1,
                retain=False,
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

        def create_error_response(self, job_id, error_type, error_message):
            """Create error response."""
            result = {
                "job_id": job_id,
                "status": "error",
                "error_type": error_type,
                "error_message": error_message,
                "failed_at": datetime.now().isoformat()
            }

            return ReturnType(
                data=result,
                topic="tasks/errors",
                qos=1,
                retain=True,  # Retain error messages
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

    # Create processor and listener
    processor = ResilientProcessor()
    listener = EventListener(config)

    # Start processing
    logger.info("Starting resilient processor...")
    await listener.run(processor.process_message)

if __name__ == "__main__":
    asyncio.run(resilient_processor_example())
```

11.3. Real-World Use Cases

11.3.1. IoT Data Processing

```
import asyncio
import json
from datetime import datetime
from Listener import EventListener, EventListenerConfig, ReturnType

async def iot_data_processor():
    """Process IoT sensor data from MQTT."""

    config = EventListenerConfig(
        host="iot-mqtt-broker.local",
        port=1883,
        topic="sensors/+/data",
        qos=1,
        max_jobs_in_memory=10000,
        job_cleanup_interval=3600
    )

    class IoTProcessor:
        def __init__(self):
            self.sensor_thresholds = {
                "temperature": {"min": -40, "max": 80, "critical": 75},
                "humidity": {"min": 0, "max": 100, "critical": 95},
                "pressure": {"min": 900, "max": 1100, "critical": 1050}
            }

        def process_sensor_data(self, data, job_id):
            """Process IoT sensor data."""
            sensor_id = data.get('sensor_id')
            sensor_type = data.get('sensor_type')
            value = data.get('value')
            timestamp = data.get('timestamp')

            if not all([sensor_id, sensor_type, value is not None]):
                return self.create_error("missing_fields", job_id)

            # Validate and process data
            processed_data = {
                "sensor_id": sensor_id,
                "sensor_type": sensor_type,
                "value": value,
                "timestamp": timestamp,
                "processed_at": datetime.now().isoformat(),
                "job_id": job_id
            }

            # Check thresholds
```

```
            alerts = self.check_thresholds(sensor_type, value, sensor_id)
            if alerts:
                processed_data["alerts"] = alerts
                self.send_alerts(alerts, sensor_id, job_id)

            # Store processed data
            return ReturnType(
                data=processed_data,
                topic=f"processed/sensors/{sensor_type}",
                qos=1,
                retain=False,
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

        def check_thresholds(self, sensor_type, value, sensor_id):
            """Check if sensor value exceeds thresholds."""
            alerts = []
            thresholds = self.sensor_thresholds.get(sensor_type, {})

            if value < thresholds.get("min", float('-inf')):
                alerts.append({
                    "type": "below_minimum",
                    "threshold": thresholds["min"],
                    "value": value
                })

            if value > thresholds.get("max", float('inf')):
                alerts.append({
                    "type": "above_maximum",
                    "threshold": thresholds["max"],
                    "value": value
                })

            if value > thresholds.get("critical", float('inf')):
                alerts.append({
                    "type": "critical",
                    "threshold": thresholds["critical"],
                    "value": value
                })

            return alerts

        def send_alerts(self, alerts, sensor_id, job_id):
            """Send alert notifications."""
            for alert in alerts:
                alert_data = {
                    "alert_id": f"alert-{job_id}-{len(alerts)}",
                    "sensor_id": sensor_id,
                    "alert_type": alert["type"],
```

```
                    "value": alert["value"],
                    "threshold": alert["threshold"],
                    "severity": "critical" if alert["type"] == "critical" else
"warning",
                    "created_at": datetime.now().isoformat()
                }

                # Would send to alerts topic
                print(f"🚨 Alert: {alert_data}")

        def create_error(self, error_type, job_id):
            """Create error response."""
            return ReturnType(
                data={
                    "job_id": job_id,
                    "status": "error",
                    "error_type": error_type,
                    "error_time": datetime.now().isoformat()
                },
                topic="sensors/errors",
                qos=1,
                retain=True,
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

    processor = IoTProcessor()
    listener = EventListener(config)

    print("Starting IoT data processor...")
    await listener.run(processor.process_sensor_data)

if __name__ == "__main__":
    asyncio.run(iot_data_processor())
```

11.3.2. Distributed Task Processing

```
import asyncio
import uuid
from datetime import datetime, timedelta
from Listener import EventListener, EventListenerConfig, ReturnType

async def distributed_task_processor():
    """Distributed task processing system."""

    config = EventListenerConfig(
        host="task-queue.example.com",
        port=1883,
```

```
        topic="tasks/queue",
        client_id=f"worker-{uuid.uuid4().hex[:8]}",
        qos=2,  # Exactly once delivery
        max_jobs_in_memory=500,
        job_cleanup_interval=1800
    )

    class TaskWorker:
        def __init__(self, worker_id):
            self.worker_id = worker_id
            self.processed_count = 0
            self.start_time = datetime.now()

        def process_task(self, data, job_id):
            """Process distributed task."""
            task_type = data.get('task_type')
            priority = data.get('priority', 'normal')
            created_at = data.get('created_at')

            # Calculate task age
            if created_at:
                task_age = datetime.now() - datetime.fromisoformat(created_at)
                if task_age > timedelta(hours=1):
                    return self.create_expired_response(job_id, task_age)

            # Process based on task type
            if task_type == 'image_processing':
                return self.process_image_task(data, job_id)
            elif task_type == 'data_analysis':
                return self.process_analysis_task(data, job_id)
            elif task_type == 'report_generation':
                return self.process_report_task(data, job_id)
            else:
                return self.create_unknown_task_response(job_id, task_type)

        def process_image_task(self, data, job_id):
            """Process image processing task."""
            image_url = data.get('image_url')
            operations = data.get('operations', [])

            # Simulate image processing
            result = {
                "job_id": job_id,
                "task_type": "image_processing",
                "worker_id": self.worker_id,
                "image_url": image_url,
                "operations_performed": operations,
                "output_url": f"processed/{job_id}.jpg",
                "processing_time": "2.3s",
                "status": "completed"
            }
```

```
            self.processed_count += 1

            return ReturnType(
                data=result,
                topic="tasks/completed/image",
                qos=2,
                retain=False,
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

        def process_analysis_task(self, data, job_id):
            """Process data analysis task."""
            dataset_id = data.get('dataset_id')
            analysis_type = data.get('analysis_type')

            # Simulate analysis
            result = {
                "job_id": job_id,
                "task_type": "data_analysis",
                "worker_id": self.worker_id,
                "dataset_id": dataset_id,
                "analysis_type": analysis_type,
                "results": {
                    "total_records": 10000,
                    "anomalies_detected": 23,
                    "confidence_score": 0.94
                },
                "status": "completed"
            }

            self.processed_count += 1

            return ReturnType(
                data=result,
                topic="tasks/completed/analysis",
                qos=2,
                retain=False,
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

        def process_report_task(self, data, job_id):
            """Process report generation task."""
            report_type = data.get('report_type')
            parameters = data.get('parameters', {})

            # Simulate report generation
```

```
            result = {
                "job_id": job_id,
                "task_type": "report_generation",
                "worker_id": self.worker_id,
                "report_type": report_type,
                "report_url": f"reports/{job_id}.pdf",
                "parameters": parameters,
                "page_count": 15,
                "status": "completed"
            }

            self.processed_count += 1

            return ReturnType(
                data=result,
                topic="tasks/completed/reports",
                qos=2,
                retain=False,
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

        def create_expired_response(self, job_id, task_age):
            """Handle expired tasks."""
            return ReturnType(
                data={
                    "job_id": job_id,
                    "status": "expired",
                    "worker_id": self.worker_id,
                    "task_age_seconds": task_age.total_seconds(),
                    "expired_at": datetime.now().isoformat()
                },
                topic="tasks/expired",
                qos=1,
                retain=False,
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

        def create_unknown_task_response(self, job_id, task_type):
            """Handle unknown task types."""
            return ReturnType(
                data={
                    "job_id": job_id,
                    "status": "error",
                    "error_type": "unknown_task_type",
                    "task_type": task_type,
                    "worker_id": self.worker_id,
                    "error_time": datetime.now().isoformat()
```

```
                },
                topic="tasks/errors",
                qos=1,
                retain=True,
                message_id=int(time.time()),
                timestamp=datetime.now(),
                job_id=job_id
            )

        def get_worker_stats(self):
            """Get worker statistics."""
            uptime = datetime.now() - self.start_time
            return {
                "worker_id": self.worker_id,
                "processed_count": self.processed_count,
                "uptime_seconds": uptime.total_seconds(),
                "rate_per_hour": self.processed_count / (uptime.total_seconds() /
3600)
            }

    # Create worker and listener
    worker_id = f"worker-{uuid.uuid4().hex[:8]}"
    worker = TaskWorker(worker_id)
    listener = EventListener(config)

    print(f"Starting distributed task worker: {worker_id}")

    # Periodically log statistics
    async def log_stats():
        while True:
            await asyncio.sleep(300)  # Every 5 minutes
            stats = worker.get_worker_stats()
            print(f"Worker stats: {stats}")

    # Start both the listener and stats logging
    stats_task = asyncio.create_task(log_stats())

    try:
        await listener.run(worker.process_task)
    finally:
        stats_task.cancel()

if __name__ == "__main__":
    asyncio.run(distributed_task_processor())
```

11.4. Testing Examples

11.4.1. Unit Test Example

```
import pytest
import asyncio
from unittest.mock import AsyncMock
from Listener import EventListener, EventListenerConfig, JobStatus

@pytest.mark.unit
@pytest.mark.asyncio
class TestExampleProcessor:
    """Example unit tests for custom processor."""

    async def test_simple_message_processing(self):
        """Test basic message processing functionality."""
        # Setup
        config = EventListenerConfig(
            host="test-host",
            port=1883,
            topic="test/topic"
        )
        listener = EventListener(config)

        # Mock MQTT client
        listener.client = AsyncMock()

        # Test data
        test_data = {
            "job_id": "test-001",
            "task_type": "test_task",
            "data": {"key": "value"}
        }

        # Define test processor
        def test_processor(data, job_id):
            return ReturnType(
                data={"processed": True, "job_id": job_id},
                topic="test/results",
                qos=0,
                retain=False,
                message_id=1,
                timestamp=datetime.now(),
                job_id=job_id
            )

        # Process message
        await listener._process_message(test_data)

        # Verify job was created and processed
        job_info = await listener.get_job_status("test-001")
        assert job_info is not None
```

```
        assert job_info.job_id == "test-001"

    async def test_error_handling(self):
        """Test error handling in processor."""
        config = EventListenerConfig()
        listener = EventListener(config)
        listener.client = AsyncMock()

        def error_processor(data, job_id):
            if data.get("should_fail"):
                raise ValueError("Test error")
            return ReturnType(
                data={"success": True},
                topic="test/results",
                qos=0,
                retain=False,
                message_id=1,
                timestamp=datetime.now(),
                job_id=job_id
            )

        # Test error condition
        error_data = {"job_id": "error-001", "should_fail": True}
        await listener._process_message(error_data)

        # Verify job failed
        job_info = await listener.get_job_status("error-001")
        assert job_info.status == JobStatus.FAILED
        assert "Test error" in job_info.error
```

11.5. Configuration Examples

11.5.1. Production Configuration

```
"""Production configuration example."""

import os
from Listener import EventListenerConfig

def get_production_config():
    """Get production-ready configuration."""
    return EventListenerConfig(
        # Connection settings
        host=os.getenv("MQTT_HOST", "prod-mqtt.example.com"),
        port=int(os.getenv("MQTT_PORT", 8883)),
        username=os.getenv("MQTT_USERNAME"),
        password=os.getenv("MQTT_PASSWORD"),
        client_id=f"prod-listener-{os.getenv('HOSTNAME', 'unknown')}",
```

```
        # SSL/TLS settings
        cafile="/etc/ssl/certs/ca-certificates.crt",

        # Connection reliability
        auto_reconnect=True,
        reconnect_retries=10,
        reconnect_max_interval=60,
        keep_alive=60,

        # Topic settings
        topic="production/events",
        qos=1,
        error_topic="production/errors",
        results_topic="production/results",

        # Job management
        max_jobs_in_memory=10000,
        job_cleanup_interval=3600,  # 1 hour
        duplicate_action="reprocess",

        # Will message for monitoring
        will={
            "topic": "production/status",
            "message": "listener_offline",
            "qos": 1,
            "retain": True
        }
    )
```

11.5.2. Development Configuration

```
"""Development configuration example."""

from Listener import EventListenerConfig

def get_development_config():
    """Get development configuration."""
    return EventListenerConfig(
        # Local development settings
        host="localhost",
        port=1883,
        username="dev",
        password="dev",
        client_id="dev-listener",

        # Development topics
        topic="dev/events",
        error_topic="dev/errors",
        results_topic="dev/results",
```

```
        # Faster feedback for development
        max_jobs_in_memory=1000,
        job_cleanup_interval=300,  # 5 minutes
        auto_reconnect=True,
        reconnect_retries=3,

        # Debug settings
        qos=0,  # Faster delivery
        retain=False
    )
```

12. Performance Guide

Performance characteristics and optimization guidelines for the MQTT Event Listener.

12.1. Performance Metrics

12.1.1. Baseline Performance

Metric	Description	Typical Value	Target
Throughput	Messages processed per second	100-1000 msg/s	>100 msg/s
Latency	Message processing delay	<100ms	<500ms
Memory Usage	RAM consumption per 1000 jobs	10-50 MB	<100 MB
CPU Usage	Processor utilization	5-15%	<25%
Connection Recovery	Time to reconnect	1-10 seconds	<30 seconds

12.1.2. Performance Factors

Message Size Impact

Message Size	Processing Time	Memory Impact	Throughput
Small (<1KB)	<10ms	~1KB per job	>500 msg/s
Medium (1-10KB)	10-50ms	~5KB per job	200-500 msg/s
Large (10-100KB)	50-200ms	~50KB per job	50-200 msg/s
Very Large (>100KB)	>200ms	>100KB per job	<50 msg/s

Processing Complexity

- **Simple operations** (data transformation) - <50ms
- **I/O operations** (file access, API calls) - 100-1000ms

- **Complex calculations** (data analysis) - 500-5000ms
- **External dependencies** (databases, services) - Variable

12.2. Optimization Strategies

12.2.1. Configuration Optimization

Memory Management

```
# Optimized for high throughput
config = EventListenerConfig(
    max_jobs_in_memory=1000,      # Reduce memory usage
    job_cleanup_interval=300,     # More frequent cleanup (5 min)
    duplicate_action="skip"       # Avoid reprocessing
)

# Optimized for reliability
config = EventListenerConfig(
    max_jobs_in_memory=10000,     # Higher job retention
    job_cleanup_interval=3600,    # Less frequent cleanup (1 hour)
    duplicate_action="reprocess"  # Handle all messages
)
```

MQTT Settings

```
# High performance settings
config = EventListenerConfig(
    qos=0,                        # Fastest delivery
    keep_alive=30,                # Shorter keepalive
    reconnect_retries=5,          # Quick reconnection
    reconnect_max_interval=30
)

# High reliability settings
config = EventListenerConfig(
    qos=2,                        # Guaranteed delivery
    keep_alive=60,                # Stable keepalive
    reconnect_retries=10,         # Persistent reconnection
    reconnect_max_interval=60
)
```

12.2.2. Processing Optimization

Async Processing

```
import asyncio
```

```
import aiohttp
from Listener import EventListener, ReturnType

async def optimized_async_processor(data, job_id):
    """High-performance async processor."""

    # Use connection pooling
    async with aiohttp.ClientSession() as session:
        # Parallel API calls
        tasks = [
            fetch_data(session, data['endpoint1']),
            fetch_data(session, data['endpoint2']),
            fetch_data(session, data['endpoint3'])
        ]

        results = await asyncio.gather(*tasks, return_exceptions=True)

    # Process results
    processed_data = combine_results(results)

    return ReturnType(
        data=processed_data,
        topic="results/optimized",
        qos=0,
        retain=False,
        message_id=int(time.time()),
        timestamp=datetime.now(),
        job_id=job_id
    )

async def fetch_data(session, endpoint):
    """Optimized data fetching with timeout."""
    timeout = aiohttp.ClientTimeout(total=5)
    async with session.get(endpoint, timeout=timeout) as response:
        return await response.json()
```

Batch Processing

```
class BatchProcessor:
    """High-throughput batch processor."""

    def __init__(self, batch_size=50, flush_interval=10.0):
        self.batch_size = batch_size
        self.flush_interval = flush_interval
        self.batch = []
        self.last_flush = time.time()

    async def process_message(self, data, job_id):
        """Add to batch and process when ready."""
        self.batch.append((data, job_id))
```

```
        # Process batch if conditions met
        if (len(self.batch) >= self.batch_size or
            time.time() - self.last_flush > self.flush_interval):
            await self.flush_batch()

    async def flush_batch(self):
        """Process entire batch efficiently."""
        if not self.batch:
            return

        batch_data = [item[0] for item in self.batch]
        job_ids = [item[1] for item in self.batch]

        # Bulk processing
        results = await process_batch_efficiently(batch_data)

        # Publish results
        for result, job_id in zip(results, job_ids):
            await publish_result(result, job_id)

        # Reset batch
        self.batch = []
        self.last_flush = time.time()
```

Caching Strategy

```
import functools
import time
from typing import Dict, Any

class CachedProcessor:
    """Processor with intelligent caching."""

    def __init__(self, cache_ttl=300):  # 5 minutes
        self.cache: Dict[str, tuple] = {}
        self.cache_ttl = cache_ttl

    def process_with_cache(self, data: Dict[str, Any], job_id: str):
        """Process with caching for expensive operations."""

        # Create cache key from relevant data
        cache_key = self.create_cache_key(data)

        # Check cache
        cached_result = self.get_cached_result(cache_key)
        if cached_result:
            return self.create_cached_response(cached_result, job_id)

        # Process and cache
```

```
        result = self.expensive_processing(data)
        self.cache_result(cache_key, result)

        return self.create_response(result, job_id)

    def create_cache_key(self, data: Dict[str, Any]) -> str:
        """Create deterministic cache key."""
        relevant_fields = ['input_data', 'parameters', 'operation_type']
        key_data = {k: data.get(k) for k in relevant_fields if k in data}
        return hashlib.md5(str(sorted(key_data.items())).encode()).hexdigest()

    def get_cached_result(self, cache_key: str):
        """Retrieve from cache if valid."""
        if cache_key in self.cache:
            result, timestamp = self.cache[cache_key]
            if time.time() - timestamp < self.cache_ttl:
                return result
            else:
                del self.cache[cache_key]  # Expired
        return None

    def cache_result(self, cache_key: str, result):
        """Store in cache with timestamp."""
        self.cache[cache_key] = (result, time.time())

        # Cleanup old entries periodically
        if len(self.cache) > 1000:
            self.cleanup_cache()

    def cleanup_cache(self):
        """Remove expired cache entries."""
        current_time = time.time()
        expired_keys = [
            key for key, (_, timestamp) in self.cache.items()
            if current_time - timestamp > self.cache_ttl
        ]
        for key in expired_keys:
            del self.cache[key]
```

12.2.3. System-Level Optimization

Memory Profiling

```
import psutil
import gc
import asyncio

class PerformanceMonitor:
    """Monitor system performance."""
```

```
    def __init__(self, listener):
        self.listener = listener
        self.stats = {
            'messages_processed': 0,
            'processing_times': [],
            'memory_samples': [],
            'error_count': 0
        }

    async def monitor_performance(self):
        """Continuous performance monitoring."""
        while True:
            await self.collect_metrics()
            await asyncio.sleep(60)  # Every minute

    async def collect_metrics(self):
        """Collect performance metrics."""
        # Memory usage
        process = psutil.Process()
        memory_mb = process.memory_info().rss / 1024 / 1024
        self.stats['memory_samples'].append(memory_mb)

        # Job statistics
        all_jobs = await self.listener.get_all_jobs()
        running_jobs = await self.listener.get_running_jobs()

        # Performance report
        if len(self.stats['memory_samples']) % 10 == 0:  # Every 10 minutes
            await self.generate_report()

    async def generate_report(self):
        """Generate performance report."""
        memory_avg = sum(self.stats['memory_samples'][-10:]) / 10
        processing_avg = sum(self.stats['processing_times'][-100:]) / 100 if
self.stats['processing_times'] else 0

        print(f"Performance Report:")
        print(f"  Average Memory: {memory_avg:.1f} MB")
        print(f"  Average Processing Time: {processing_avg:.3f}s")
        print(f"  Messages Processed: {self.stats['messages_processed']}")
        print(f"  Error Rate: {self.stats['error_count'] / max(1,
self.stats['messages_processed']) * 100:.1f}%")
```

Resource Management

```
import resource
import signal

def setup_resource_limits():
```

```
    """Configure system resource limits."""

    # Memory limit (1GB)
    resource.setrlimit(resource.RLIMIT_AS, (1024 * 1024 * 1024, -1))

    # File descriptor limit
    resource.setrlimit(resource.RLIMIT_NOFILE, (1024, 4096))

    # CPU time limit (if needed)
    # resource.setrlimit(resource.RLIMIT_CPU, (3600, -1))  # 1 hour

def setup_graceful_shutdown(listener):
    """Setup graceful shutdown handling."""

    def signal_handler(signum, frame):
        print(f"Received signal {signum}, initiating graceful shutdown...")
        listener.stop()

    signal.signal(signal.SIGINT, signal_handler)
    signal.signal(signal.SIGTERM, signal_handler)
```

12.3. Performance Monitoring

12.3.1. Built-in Metrics

```
class MetricsCollector:
    """Collect performance metrics during operation."""

    def __init__(self):
        self.start_time = time.time()
        self.message_count = 0
        self.processing_times = []
        self.error_count = 0

    def record_processing_time(self, duration: float):
        """Record message processing time."""
        self.processing_times.append(duration)
        self.message_count += 1

        # Keep only recent measurements
        if len(self.processing_times) > 1000:
            self.processing_times = self.processing_times[-500:]

    def record_error(self):
        """Record processing error."""
        self.error_count += 1

    def get_metrics(self) -> Dict[str, Any]:
        """Get current performance metrics."""
```

```
        uptime = time.time() - self.start_time

        if self.processing_times:
            avg_time = sum(self.processing_times) / len(self.processing_times)
            p95_time = sorted(self.processing_times)[int(len(self.processing_times) *
0.95)]
        else:
            avg_time = p95_time = 0

        return {
            'uptime_seconds': uptime,
            'messages_processed': self.message_count,
            'messages_per_second': self.message_count / uptime if uptime > 0 else 0,
            'average_processing_time': avg_time,
            'p95_processing_time': p95_time,
            'error_count': self.error_count,
            'error_rate': self.error_count / max(1, self.message_count)
        }

# Integration with EventListener
def instrumented_processor(metrics: MetricsCollector):
    """Create instrumented processor."""

    def processor(data, job_id):
        start_time = time.time()

        try:
            result = your_processing_logic(data, job_id)
            return result
        except Exception as e:
            metrics.record_error()
            raise
        finally:
            processing_time = time.time() - start_time
            metrics.record_processing_time(processing_time)

    return processor
```

12.3.2. Benchmarking

```
import asyncio
import time
from typing import List

async def benchmark_throughput(listener, message_count=1000):
    """Benchmark message processing throughput."""

    processed_jobs = []
```

```
    def benchmark_processor(data, job_id):
        """Simple processor for benchmarking."""
        processed_jobs.append(job_id)
        return ReturnType(
            data={"processed": True, "job_id": job_id},
            topic="benchmark/results",
            qos=0,
            retain=False,
            message_id=1,
            timestamp=datetime.now(),
            job_id=job_id
        )

    # Generate test messages
    test_messages = [
        {"job_id": f"benchmark-{i}", "data": f"test-data-{i}"}
        for i in range(message_count)
    ]

    # Measure processing time
    start_time = time.time()

    # Process messages
    for message in test_messages:
        await listener._process_message(message)

    # Wait for completion
    while len(processed_jobs) < message_count:
        await asyncio.sleep(0.1)

    end_time = time.time()
    duration = end_time - start_time
    throughput = message_count / duration

    print(f"Benchmark Results:")
    print(f"  Messages: {message_count}")
    print(f"  Duration: {duration:.2f} seconds")
    print(f"  Throughput: {throughput:.1f} messages/second")

    return throughput

async def benchmark_memory_usage(listener, job_count=10000):
    """Benchmark memory usage with many jobs."""

    import psutil
    process = psutil.Process()

    initial_memory = process.memory_info().rss / 1024 / 1024

    # Create many jobs
    for i in range(job_count):
```

```
        job_data = {"job_id": f"memory-test-{i}", "data": f"data-{i}"}
        await listener._create_job(f"memory-test-{i}", job_data)

    peak_memory = process.memory_info().rss / 1024 / 1024
    memory_per_job = (peak_memory - initial_memory) / job_count * 1024  # KB per job

    print(f"Memory Benchmark:")
    print(f"  Jobs Created: {job_count}")
    print(f"  Initial Memory: {initial_memory:.1f} MB")
    print(f"  Peak Memory: {peak_memory:.1f} MB")
    print(f"  Memory per Job: {memory_per_job:.2f} KB")

    return memory_per_job
```

12.4. Performance Tuning Guidelines

12.4.1. For High Throughput

1. **Use QoS 0** for maximum speed
2. **Reduce job retention** time and count
3. **Implement batch processing** for multiple messages
4. **Use async operations** for I/O
5. **Minimize logging** in production
6. **Optimize processing function** complexity

12.4.2. For High Reliability

1. **Use QoS 1 or 2** for guaranteed delivery
2. **Increase job retention** for audit trails
3. **Implement retry mechanisms** in processing
4. **Use comprehensive error handling**
5. **Enable detailed logging** for debugging
6. **Configure appropriate timeouts**

12.4.3. For Low Latency

1. **Minimize processing** function complexity
2. **Use local resources** over network calls
3. **Implement caching** for repeated operations
4. **Reduce serialization** overhead
5. **Optimize network** connectivity to broker
6. **Use faster storage** for any file operations

12.4.4. For Low Memory Usage

1. **Reduce** `max_jobs_in_memory` setting
2. **Decrease** `job_cleanup_interval` for frequent cleanup
3. **Use** `duplicate_action="skip"` to avoid reprocessing
4. **Implement result streaming** instead of accumulation
5. **Use lazy loading** for large data structures
6. **Profile memory usage** regularly

13. Security Guide

Security considerations and best practices for the MQTT Event Listener.

13.1. Security Overview

13.1.1. Security Layers

Layer	Components	Status
Network	SSL/TLS, Authentication, Authorization	✅ Implemented
Application	Input validation, Error handling	✅ Implemented
Data	Configuration protection, Memory safety	✅ Implemented
Process	Resource limits, Secure logging	✅ Implemented

13.2. Network Security

13.2.1. SSL/TLS Configuration

```
# Secure SSL configuration
config = EventListenerConfig(
    host="secure-mqtt.example.com",
    port=8883,  # SSL port

    # Certificate validation
    cafile="/etc/ssl/certs/ca.crt",
    capath="/etc/ssl/certs/",

    # Authentication
    username=os.getenv("MQTT_USERNAME"),
    password=os.getenv("MQTT_PASSWORD")
)
```

13.2.2. Authentication Best Practices

- **Use environment variables** for credentials
- **Rotate passwords** regularly
- **Use strong authentication** (certificates when possible)
- **Limit connection privileges** to minimum required

```
# Secure credential handling
import os
from getpass import getpass

def get_secure_config():
    return EventListenerConfig(
        username=os.getenv("MQTT_USER"),
        password=os.getenv("MQTT_PASS") or getpass("MQTT Password: "),
        cafile=os.getenv("MQTT_CA_FILE", "/etc/ssl/ca.crt")
    )
```

13.3. Application Security

13.3.1. Input Validation

- **TOML syntax validation** prevents injection attacks
- **Message size limits** prevent DoS attacks
- **Configuration validation** ensures safe operation

```
# Secure message processing
def secure_processor(data, job_id):
    # Validate input size
    if len(str(data)) > 1024 * 1024:  # 1MB limit
        raise ValueError("Message too large")

    # Validate required fields
    required = ['job_id', 'task_type']
    if not all(field in data for field in required):
        raise ValueError("Missing required fields")

    # Sanitize data
    sanitized_data = sanitize_input(data)

    return process_safely(sanitized_data, job_id)
```

13.3.2. Error Handling Security

- **No sensitive data** in error messages

- **Sanitized logging** to prevent information disclosure
- **Controlled error responses**

```
def secure_error_handler(job_id, error):
    # Log detailed error internally
    logger.error(f"Job {job_id} failed: {error}", extra={'job_id': job_id})

    # Return sanitized error
    return ReturnType(
        data={
            "job_id": job_id,
            "status": "error",
            "error_code": "PROCESSING_ERROR",
            # No sensitive details exposed
        },
        topic="errors",
        qos=1,
        retain=False,
        message_id=int(time.time()),
        timestamp=datetime.now(),
        job_id=job_id
    )
```

13.4. Configuration Security

13.4.1. Sensitive Data Protection

```
# Secure configuration management
class SecureConfig:
    def __init__(self):
        self.sensitive_fields = {'password', 'token', 'key', 'secret'}

    def sanitize_config_for_logging(self, config_dict):
        """Remove sensitive data from logs."""
        sanitized = {}
        for key, value in config_dict.items():
            if any(sensitive in key.lower() for sensitive in self.sensitive_fields):
                sanitized[key] = "***REDACTED***"
            else:
                sanitized[key] = value
        return sanitized
```

13.4.2. Environment Security

```
# Secure environment setup
export MQTT_HOST="secure-broker.internal"
```

```
export MQTT_PORT="8883"
export MQTT_USERNAME="service-account"
export MQTT_PASSWORD="$(cat /secure/mqtt.pass)"
export MQTT_CA_FILE="/etc/ssl/internal-ca.crt"

# Restrict environment file access
chmod 600 ~/.env
```

13.5. Process Security

13.5.1. Resource Limits

```
# Security-focused configuration
config = EventListenerConfig(
    max_jobs_in_memory=1000,     # Limit memory usage
    job_cleanup_interval=300,    # Frequent cleanup
    connect_timeout=30,          # Prevent hanging connections
    reconnect_retries=5          # Limit retry attempts
)
```

13.5.2. Secure Logging

```
import logging
import re

class SecureFormatter(logging.Formatter):
    """Logging formatter that sanitizes sensitive data."""

    def __init__(self):
        super().__init__(
            '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        )
        # Patterns to redact
        self.patterns = [
            (re.compile(r'password["\']?\s*[:=]\s*["\']?([^"\'\s]+)', re.I),
'password=***'),
            (re.compile(r'token["\']?\s*[:=]\s*["\']?([^"\'\s]+)', re.I),
'token=***'),
            (re.compile(r'key["\']?\s*[:=]\s*["\']?([^"\'\s]+)', re.I), 'key=***'),
        ]

    def format(self, record):
        msg = super().format(record)

        # Sanitize sensitive data
        for pattern, replacement in self.patterns:
            msg = pattern.sub(replacement, msg)
```

```
        return msg

# Configure secure logging
handler = logging.StreamHandler()
handler.setFormatter(SecureFormatter())
logger = logging.getLogger('Listener')
logger.addHandler(handler)
```

13.6. Security Monitoring

13.6.1. Threat Detection

```
class SecurityMonitor:
    def __init__(self):
        self.failed_attempts = {}
        self.suspicious_patterns = [
            'script', 'exec', 'eval', 'import',
            '../', '..\\', '/etc/', 'cmd.exe'
        ]

    def check_message_security(self, data, job_id):
        """Check message for security threats."""
        message_str = str(data).lower()

        # Check for suspicious patterns
        for pattern in self.suspicious_patterns:
            if pattern in message_str:
                self.log_security_event(
                    f"Suspicious pattern '{pattern}' in job {job_id}"
                )
                return False

        # Check message size
        if len(message_str) > 100000:  # 100KB
            self.log_security_event(f"Oversized message in job {job_id}")
            return False

        return True

    def log_security_event(self, event):
        """Log security events for monitoring."""
        logger.warning(f"SECURITY: {event}")
```

13.7. Compliance and Auditing

13.7.1. Audit Logging

```
class AuditLogger:
    def __init__(self):
        self.audit_logger = logging.getLogger('audit')

    def log_job_processing(self, job_id, status, user_context=None):
        """Log job processing for audit trail."""
        audit_data = {
            'timestamp': datetime.now().isoformat(),
            'job_id': job_id,
            'status': status,
            'user_context': user_context,
            'service': 'mqtt-event-listener'
        }

        self.audit_logger.info(json.dumps(audit_data))
```

13.8. Security Best Practices

13.8.1. Deployment Security

1. **Use SSL/TLS** for all MQTT connections
2. **Store credentials** in environment variables or secure vaults
3. **Limit network access** to MQTT broker
4. **Run with minimal privileges** (non-root user)
5. **Monitor for security events** and anomalies
6. **Keep dependencies updated** for security patches
7. **Use internal networks** when possible
8. **Implement logging** for security monitoring

13.8.2. Development Security

1. **Code review** all changes for security implications
2. **Use static analysis** tools (bandit) for vulnerability detection
3. **Test security controls** with unit tests
4. **Follow secure coding** practices
5. **Validate all inputs** from external sources
6. **Handle errors** without exposing sensitive information

13.8.3. Operational Security

1. **Regular security assessments** of the deployment
2. **Monitor logs** for suspicious activity
3. **Rotate credentials** periodically
4. **Update certificates** before expiration
5. **Backup and test** recovery procedures
6. **Document security procedures** for incident response

13.9. Security Checklist

13.9.1. Pre-Deployment

- ▢ SSL/TLS properly configured
- ▢ Credentials stored securely
- ▢ Input validation implemented
- ▢ Error handling sanitized
- ▢ Logging configured securely
- ▢ Resource limits set
- ▢ Security monitoring enabled

13.9.2. Regular Maintenance

- ▢ Dependencies updated
- ▢ Certificates renewed
- ▢ Credentials rotated
- ▢ Logs reviewed for threats
- ▢ Performance monitoring
- ▢ Security assessments

13.9.3. Incident Response

- ▢ Contact procedures documented
- ▢ Log collection automated
- ▢ Service shutdown procedures
- ▢ Recovery procedures tested
- ▢ Communication plan ready

14. Troubleshooting Guide

Common issues and solutions for the MQTT Event Listener.

14.1. Installation Issues

14.1.1. Git Authentication Errors

Problem: Cannot clone repository or install from git.

Symptoms:

```
fatal: Authentication failed
Permission denied (publickey)
```

Solutions:

1. **Check repository access:**

   ```
   git ls-remote https://github.com/ed-00/Mqtt-client.git
   ```

2. **Configure Git credentials:**

   ```
   git config --global user.name "Your Name"
   git config --global user.email "your.email@example.com"
   ```

3. **Use SSH key authentication:**

   ```
   ssh-keygen -t rsa -b 4096 -C "your.email@example.com"
   # Add public key to your Git provider
   ```

4. **Contact your administrator** for repository access

14.1.2. Python Version Issues

Problem: Installation fails due to Python version incompatibility.

Symptoms:

```
ERROR: Package requires Python '>=3.8' but the running Python is 3.7
```

Solutions:

1. **Check Python version:**

```
python --version
python3 --version
```

2. **Use Python 3.8 or later:**

```
# Install Python 3.8+ using package manager
sudo apt update && sudo apt install python3.8

# Or use pyenv
pyenv install 3.8.10
pyenv local 3.8.10
```

3. **Use virtual environment:**

```
python3.8 -m venv mqtt-env
source mqtt-env/bin/activate
pip install git+{link-repo}.git
```

14.1.3. Dependency Conflicts

Problem: Package installation fails due to dependency conflicts.

Symptoms:

```
ERROR: pip's dependency resolver does not currently consider pre-releases
ResolutionImpossible: for help visit https://pip.pypa.io/en/stable/...
```

Solutions:

1. **Use clean virtual environment:**

```
python -m venv clean-env
source clean-env/bin/activate
pip install --upgrade pip
pip install git+{link-repo}.git
```

2. **Install with no dependencies first:**

```
pip install --no-deps git+{link-repo}.git
pip install -r requirements.txt
```

3. **Check for conflicting packages:**

```
pip check
pip list --outdated
```

14.2. Connection Issues

14.2.1. MQTT Broker Connection Failures

Problem: Cannot connect to MQTT broker.

Symptoms:

```
ConnectionError: [Errno 111] Connection refused
TimeoutError: Connection timeout
```

Solutions:

1. **Verify broker connectivity:**

```
# Test network connectivity
ping mqtt-broker-host

# Test port connectivity
telnet mqtt-broker-host 1883
nc -zv mqtt-broker-host 1883
```

2. **Check configuration:**

```
config = EventListenerConfig(
    host="correct-host",
    port=1883,  # or 8883 for SSL
    username="valid-username",
    password="valid-password"
)
```

3. **Enable debug logging:**

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('amqtt.client').setLevel(logging.DEBUG)
```

4. **Test with MQTT client tools:**

```
# Install mosquitto-clients
sudo apt install mosquitto-clients
```

```
# Test connection
mosquitto_sub -h mqtt-host -p 1883 -t test/topic -u username -P password
```

14.2.2. SSL/TLS Connection Issues

Problem: SSL connection fails or certificate errors.

Symptoms:

```
SSL: CERTIFICATE_VERIFY_FAILED
ssl.SSLError: [SSL: WRONG_VERSION_NUMBER]
```

Solutions:

1. **Verify SSL configuration:**

```
config = EventListenerConfig(
    host="ssl-broker",
    port=8883,  # SSL port
    cafile="/path/to/ca.crt",
    # or
    cadata="-----BEGIN CERTIFICATE-----..."
)
```

2. **Check certificate validity:**

```
openssl s_client -connect ssl-broker:8883 -servername ssl-broker
openssl x509 -in ca.crt -text -noout
```

3. **Disable certificate verification (testing only):**

```
import ssl
config = EventListenerConfig(
    host="ssl-broker",
    port=8883,
    tls_context=ssl.create_default_context()
)
config.tls_context.check_hostname = False
config.tls_context.verify_mode = ssl.CERT_NONE
```

14.2.3. Network Timeout Issues

Problem: Connection timeouts or intermittent disconnections.

Solutions:

1. **Adjust timeout settings:**

```
config = EventListenerConfig(
    keep_alive=60,
    reconnect_retries=10,
    reconnect_max_interval=60,
    auto_reconnect=True
)
```

2. **Check network stability:**

```
# Monitor network connectivity
ping -c 10 mqtt-broker-host

# Check for packet loss
mtr mqtt-broker-host
```

3. **Use connection monitoring:**

```
import asyncio

async def monitor_connection(listener):
    while True:
        if listener.client and listener.client.is_connected():
            print("✓ Connected")
        else:
            print("✗ Disconnected")
        await asyncio.sleep(30)
```

14.3. Runtime Issues

14.3.1. Memory Issues

Problem: High memory usage or memory leaks.

Symptoms:

```
MemoryError: Unable to allocate memory
Process killed (OOM killer)
```

Solutions:

1. **Reduce job memory limit:**

```
config = EventListenerConfig(
    max_jobs_in_memory=1000,  # Reduce from default 5000
    job_cleanup_interval=600  # More frequent cleanup
)
```

2. **Monitor memory usage:**

```
import psutil

def monitor_memory():
    process = psutil.Process()
    memory_mb = process.memory_info().rss / 1024 / 1024
    print(f"Memory usage: {memory_mb:.1f} MB")

# Call periodically
asyncio.create_task(periodic_memory_check())
```

3. **Manual cleanup:**

```
# Cleanup old jobs manually
await listener.cleanup_old_jobs()

# Force garbage collection
import gc
gc.collect()
```

14.3.2. Performance Issues

Problem: Slow message processing or high latency.

Solutions:

1. **Profile message processing:**

```
import time

def timed_processor(data, job_id):
    start_time = time.time()
    result = your_processing_logic(data, job_id)
    end_time = time.time()

    processing_time = end_time - start_time
    if processing_time > 1.0:  # Log slow operations
        print(f"Slow processing for {job_id}: {processing_time:.2f}s")

    return result
```

2. **Optimize configuration:**

```
config = EventListenerConfig(
    qos=0,  # Faster delivery for non-critical messages
    duplicate_action="skip",  # Avoid reprocessing
    job_cleanup_interval=300  # More frequent cleanup
)
```

3. **Use async operations:**

```
async def async_processor(data, job_id):
    # Use async operations for I/O
    async with aiohttp.ClientSession() as session:
        result = await session.get(api_url)
    return create_return_type(result, "results", job_id)
```

14.3.3. Job Processing Errors

Problem: Jobs fail or get stuck in processing.

Solutions:

1. **Add comprehensive error handling:**

```
def robust_processor(data, job_id):
    try:
        return process_data(data, job_id)
    except ValueError as e:
        logger.error(f"Validation error for {job_id}: {e}")
        return create_error_response(job_id, "validation", str(e))
    except Exception as e:
        logger.exception(f"Unexpected error for {job_id}")
        return create_error_response(job_id, "processing", str(e))
```

2. **Monitor job status:**

```
async def monitor_jobs(listener):
    running_jobs = await listener.get_running_jobs()
    for job_id, job_info in running_jobs.items():
        age = datetime.now() - job_info.started_at
        if age.total_seconds() > 300:  # 5 minutes
            print(f"⚠️ Long-running job: {job_id}")
```

3. **Check job queue health:**

```
all_jobs = await listener.get_all_jobs()
```

```
running = await listener.get_running_jobs()
completed = await listener.get_completed_jobs()
failed = len([j for j in all_jobs.values() if j.status == JobStatus.FAILED])

print(f"Jobs - Total: {len(all_jobs)}, Running: {len(running)}, "
      f"Completed: {len(completed)}, Failed: {failed}")
```

14.4. Configuration Issues

14.4.1. Invalid Configuration

Problem: Configuration validation errors.

Symptoms:

```
ValueError: Invalid port number: -1
ConfigError: Invalid QoS value: 5
```

Solutions:

1. **Validate configuration values:**

```
# Valid port range
config = EventListenerConfig(port=1883)  # 1-65535

# Valid QoS values
config = EventListenerConfig(qos=1)  # 0, 1, or 2

# Valid host format
config = EventListenerConfig(host="mqtt.example.com")
```

2. **Check environment variables:**

```
import os

def validate_env_config():
    required_vars = ["MQTT_HOST", "MQTT_USERNAME", "MQTT_PASSWORD"]
    missing = [var for var in required_vars if not os.getenv(var)]
    if missing:
        raise ValueError(f"Missing environment variables: {missing}")
```

14.4.2. TOML Parsing Errors

Problem: Cannot parse TOML message content.

Symptoms:

```
ConfigError: Invalid TOML syntax
TOMLDecodeError: Expected '=' after key
```

Solutions:

1. **Validate TOML format:**

```
import toml

def validate_toml_message(message_str):
    try:
        data = toml.loads(message_str)
        return data
    except toml.TomlDecodeError as e:
        print(f"TOML error: {e}")
        return None
```

2. **Check message format:**

```
# Correct TOML format
job_id = "task-001"
task_type = "processing"

[data]
input = "file.txt"
output = "result.txt"
```

3. **Use SafeConfigParser:**

```
from Listener import SafeConfigParser

parser = SafeConfigParser()
try:
    data = parser.parse_config_from_string(toml_string)
except ConfigError as e:
    print(f"Parse error: {e}")
```

14.5. Debugging Tools

14.5.1. Enable Debug Logging

```
import logging

# Enable all debug logging
logging.basicConfig(
```

```
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

# Specific logger levels
logging.getLogger('Listener').setLevel(logging.DEBUG)
logging.getLogger('amqtt.client').setLevel(logging.DEBUG)
logging.getLogger('amqtt.broker').setLevel(logging.INFO)
```

14.5.2. Connection Testing

```
async def test_connection(config):
    """Test MQTT connection independently."""
    from amqtt.client import MQTTClient

    client = MQTTClient()
    try:
        await client.connect(config.uri)
        print("✅ Connection successful")
        await client.disconnect()
    except Exception as e:
        print(f"❌ Connection failed: {e}")
```

14.5.3. Message Monitoring

```
def debug_processor(data, job_id):
    """Processor with extensive debugging."""
    print(f"🔍 Received job {job_id}")
    print(f"📊 Data: {data}")
    print(f"⚙️ Processing...")

    try:
        result = your_processing_logic(data, job_id)
        print(f"✅ Job {job_id} completed")
        return result
    except Exception as e:
        print(f"❌ Job {job_id} failed: {e}")
        raise
```

14.6. Getting Help

14.6.1. Before Seeking Help

1. **Check logs** for error messages and stack traces
2. **Verify configuration** against documentation

3. **Test connectivity** to MQTT broker
4. **Check system resources** (memory, CPU, network)
5. **Review recent changes** that might have caused issues

14.6.2. Support Channels

1. **Repository Issues**: Check GitHub issues for known problems
2. **Internal Documentation**: Review Internal Distribution Guide
3. **Maintainer Contact**: Email aahameed@kth.se for direct support
4. **Team Resources**: Contact your internal team or administrator

14.6.3. Information to Include

When reporting issues, include:

- **Version information** (library version, Python version)
- **Configuration** (sanitized, remove credentials)
- **Error messages** (full stack trace)
- **Environment details** (OS, network setup)
- **Steps to reproduce** the issue
- **Expected vs actual behavior**

```
# Gather system information
python --version
pip show mqtt-event-listener
uname -a
netstat -an | grep :1883
```

15. Changelog

Version history and changes for the MQTT Event Listener.

15.1. Version 1.0.1 (2025-07-01)

Comprehensive Test Suite Overhaul

15.1.1. Major Testing Improvements

- **Test Coverage Enhancement** - Increased from ~45% to **83.72%** coverage, exceeding 80% target
- **Complete Test Suite Rewrite** - Transformed 12 failing tests to **73 passing tests** (100% success rate)
- **MQTT Integration Testing** - Added 10 new tests against real Mosquitto broker

- **EventListener Focus** - Corrected tests to properly validate EventListener class instead of raw MQTT clients
- **Pytest Configuration Fix** - Fixed pytest.ini header from `[tool:pytest]` to `[pytest]` resolving marker warnings

15.1.2. Test Architecture Improvements

- **Test Organization** - Restructured into clear categories:
  - **48 Unit Tests** - Individual component validation
  - **15 Integration Tests** - Component interaction with mocks
  - **10 MQTT Integration Tests** - Real broker communication
- **Comprehensive Fixtures** - Added authenticated MQTT fixtures for real broker testing
- **Job Lifecycle Testing** - Complete job management validation from creation to cleanup
- **Error Handling Tests** - Comprehensive error condition and edge case coverage

15.1.3. EventListener Implementation Fixes

- **Message Encoding Fix** - Fixed `_send_message()` to properly encode strings to bytes for MQTT client
- **Result Publishing** - Fixed result serialization to JSON before MQTT publishing
- **Type Safety** - Enhanced type checking throughout EventListener implementation
- **Authentication Support** - Proper credential handling in MQTT integration tests

15.1.4. Testing Framework Enhancements

- **TOML Message Processing** - Comprehensive testing of TOML-formatted message handling
- **Job Tracking Validation** - End-to-end job status and lifecycle testing
- **Memory Management Tests** - Job cleanup and memory limit validation
- **Performance Benchmarks** - Test execution time optimization (under 20 seconds)

15.1.5. Quality Metrics Achieved

- **Code Coverage**: 83.72% (EventListener: 80%, SafeConfigParser: 90%)
- **Test Success Rate**: 100% (73/73 tests passing)
- **MQTT Integration**: 100% success with real Mosquitto broker
- **Documentation**: Updated test documentation with new patterns and examples

15.2. Version 1.0.0 (2025-01-09)

🎉 Initial Internal Release

15.2.1. Features

- **MQTT Event Listener** - Asynchronous MQTT client with comprehensive job tracking
- **TOML Message Processing** - Automatic parsing and validation of TOML message content
- **Job Management System** - In-memory job tracking with status monitoring
- **Configuration Management** - Unified configuration system with validation
- **Safe Configuration Parser** - Robust TOML parsing with error handling
- **Duplicate Detection** - Configurable duplicate job handling
- **Error Handling** - Comprehensive error management and reporting
- **Type Safety** - Full type hints for better IDE support

15.2.2. Core Components

- `EventListener` - Main listener class with job tracking capabilities
- `EventListenerConfig` - Configuration dataclass with validation
- `SafeConfigParser` - TOML parser with safe error handling
- `JobInfo` - Job information tracking with status and metadata
- `ReturnType` - Structured return type for processed messages
- `JobStatus` - Enumeration for job execution states

15.2.3. Job Tracking Features

- **In-memory job management** with configurable limits
- **Job status tracking** (pending, running, completed, failed, duplicate)
- **Automatic cleanup** of old jobs based on configurable intervals
- **Job querying** by status and job ID
- **Duplicate handling** with configurable actions (skip, reprocess, error)

15.2.4. MQTT Features

- **Auto-reconnection** with exponential backoff
- **SSL/TLS support** with certificate validation
- **QoS levels** 0, 1, and 2 support
- **Topic wildcards** support (`+` and `#`)
- **Will messages** for connection status monitoring
- **Custom topic configurations** with individual settings

15.2.5. Configuration Features

- **Environment variable** integration

- **File-based configuration** support
- **Validation** with meaningful error messages
- **SSL/TLS configuration** options
- **Connection reliability** settings
- **Job management** parameters

15.2.6. Development Features

- **Comprehensive testing** with 48 unit tests and 15 integration tests
- **83.72% code coverage** exceeding 80% target
- **Type hints** for all public APIs
- **Documentation** with usage examples
- **Code quality** tools (flake8, bandit, safety)

15.2.7. Distribution Features

- **Modern Python packaging** with `pyproject.toml`
- **Git-based installation** for internal use
- **Wheel distribution** support
- **Development dependencies** for contributors
- **Build automation** with scripts
- **Internal distribution** guide

15.2.8. Documentation

- **Comprehensive README** with installation and usage
- **API reference** with complete method documentation
- **Configuration guide** with all options explained
- **Usage examples** for common scenarios
- **Development guide** for contributors
- **Testing framework** documentation
- **Troubleshooting guide** for common issues

15.2.9. Technical Details

- **Python 3.8+** compatibility
- **Asyncio-based** for high performance
- **Memory efficient** with configurable job limits
- **Thread-safe** job management

- **Graceful shutdown** handling

15.3. Upcoming Features

15.3.1. Version 1.1.0 (Planned)

Planned features for the next release:

Enhanced Job Management

- **Persistent job storage** option for reliability
- **Job priority queues** for different processing priorities
- **Job scheduling** with delayed execution
- **Batch processing** capabilities for improved throughput
- **Job retry mechanisms** with exponential backoff

Monitoring and Observability

- **Metrics collection** (Prometheus/StatsD support)
- **Health check endpoints** for monitoring
- **Performance dashboards** integration
- **Distributed tracing** support
- **Custom alerting** based on job failures

Advanced Configuration

- **Dynamic configuration** reload without restart
- **Configuration profiles** for different environments
- **Secret management** integration
- **Configuration validation** schemas
- **Hot-swappable processors** for A/B testing

Scalability Improvements

- **Horizontal scaling** support
- **Load balancing** across multiple instances
- **Cluster coordination** for distributed processing
- **Message partitioning** strategies
- **Automatic scaling** based on queue depth

Developer Experience

- **Plugin system** for extensibility

- **Custom serializers** beyond TOML
- **Message transformation** pipelines
- **Development mode** with enhanced debugging
- **Configuration wizard** for easy setup

15.4. Version History Guidelines

15.4.1. Semantic Versioning

The project follows semantic versioning (SemVer):

- **MAJOR** version for incompatible API changes
- **MINOR** version for backwards-compatible functionality additions
- **PATCH** version for backwards-compatible bug fixes

15.4.2. Release Types

Major Releases (x.0.0)

- Breaking API changes
- Major architecture updates
- Significant new features
- Migration guides provided

Minor Releases (x.y.0)

- New features and capabilities
- Performance improvements
- New configuration options
- Backwards compatibility maintained

Patch Releases (x.y.z)

- Bug fixes
- Security updates
- Documentation improvements
- Performance optimizations

15.4.3. Change Categories

Changes are categorized as:

- **Features** - New functionality and capabilities

- **Improvements** - Enhancements to existing features
- **Bug Fixes** - Resolution of reported issues
- **Security** - Security-related updates
- **Documentation** - Documentation updates and improvements
- **Development** - Changes affecting developers and contributors
- **Infrastructure** - Build, CI/CD, and deployment changes

15.5. Migration Guidelines

15.5.1. Future API Changes

Guidelines for handling future breaking changes:

Deprecation Policy

1. **Advance notice** - Deprecated features announced in minor releases
2. **Deprecation period** - Minimum 2 minor versions before removal
3. **Migration guides** - Detailed guides for breaking changes
4. **Backwards compatibility** - Maintained during deprecation period

Migration Support

1. **Automated migration tools** when possible
2. **Step-by-step guides** for manual migration
3. **Example code** showing old vs new approaches
4. **Testing strategies** for validation
5. **Support channels** for migration assistance

15.6. Contributing to Changelog

15.6.1. For Maintainers

When releasing new versions:

1. **Update version numbers** in all relevant files
2. **Document all changes** in this changelog
3. **Create git tags** for releases
4. **Update internal distribution** documentation
5. **Notify team** of new releases

15.6.2. Change Documentation Format

```
=== Version X.Y.Z (YYYY-MM-DD)

*Brief description*

==== Features
* New feature description
* Another new feature

==== Improvements
* Enhancement description
* Performance improvement

==== Bug Fixes
* Bug fix description
* Security fix description

==== Breaking Changes
* Breaking change with migration guide
```

15.6.3. Internal Release Notes

For each release, maintain internal notes including:

- **Performance benchmarks** and comparisons
- **Known issues** and workarounds
- **Deployment considerations** for internal systems
- **Testing coverage** reports
- **Security assessment** results

15.7. Support Information

15.7.1. Version Support

- **Current version** (1.0.0) - Full support
- **Previous minor** (when available) - Security fixes only
- **Older versions** - Community support through issues

15.7.2. End of Life Policy

- **Major versions** - Supported for 2 years after release
- **Minor versions** - Supported until next major release
- **Security fixes** - Backported to supported versions only

15.7.3. Upgrade Recommendations

- **Stay current** with latest minor version for new features
- **Plan upgrades** around major releases for breaking changes
- **Test thoroughly** before upgrading production systems
- **Review changelogs** before upgrading

For questions about specific versions or upgrade paths, contact aahameed@kth.se.

16. Additional Resources

16.1. Internal Resources

- Internal Distribution Guide
- Project Repository
- Issue Tracker
- License

16.2. External Resources

- MQTT Protocol
- TOML Format
- Python asyncio
- pytest Testing Framework

17. Support and Contact

For questions, issues, or contributions:

- **Email**: aahameed@kth.se
- **Issues**: https://github.com/ed-00/Mqtt-client/issues
- **Internal Team**: Contact your administrator

18. License

This project is licensed under the Apache 2.0 License - see the LICENSE file for details.

Generated on 2025-06-30 for version 1.0.0