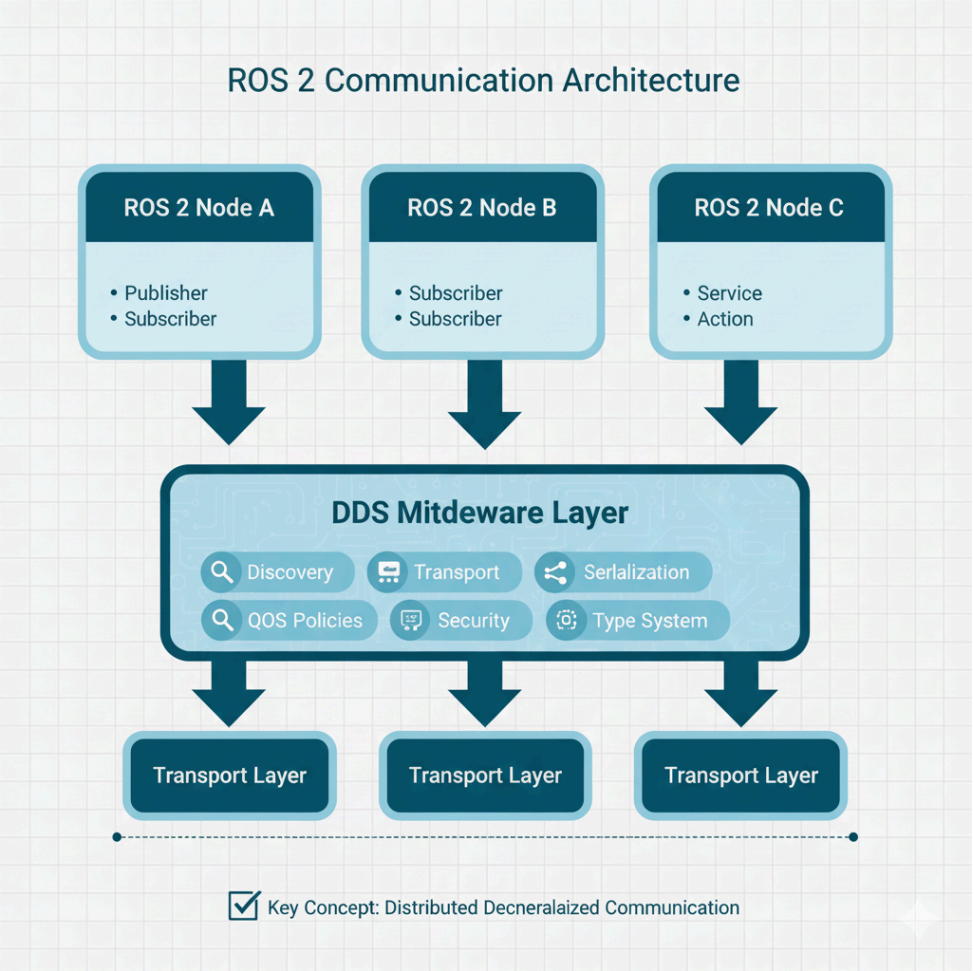
ROS 2 Communication & Middleware - Complete Training Document

1. Introduction to ROS 2 Middleware

What is ROS 2 Middleware?

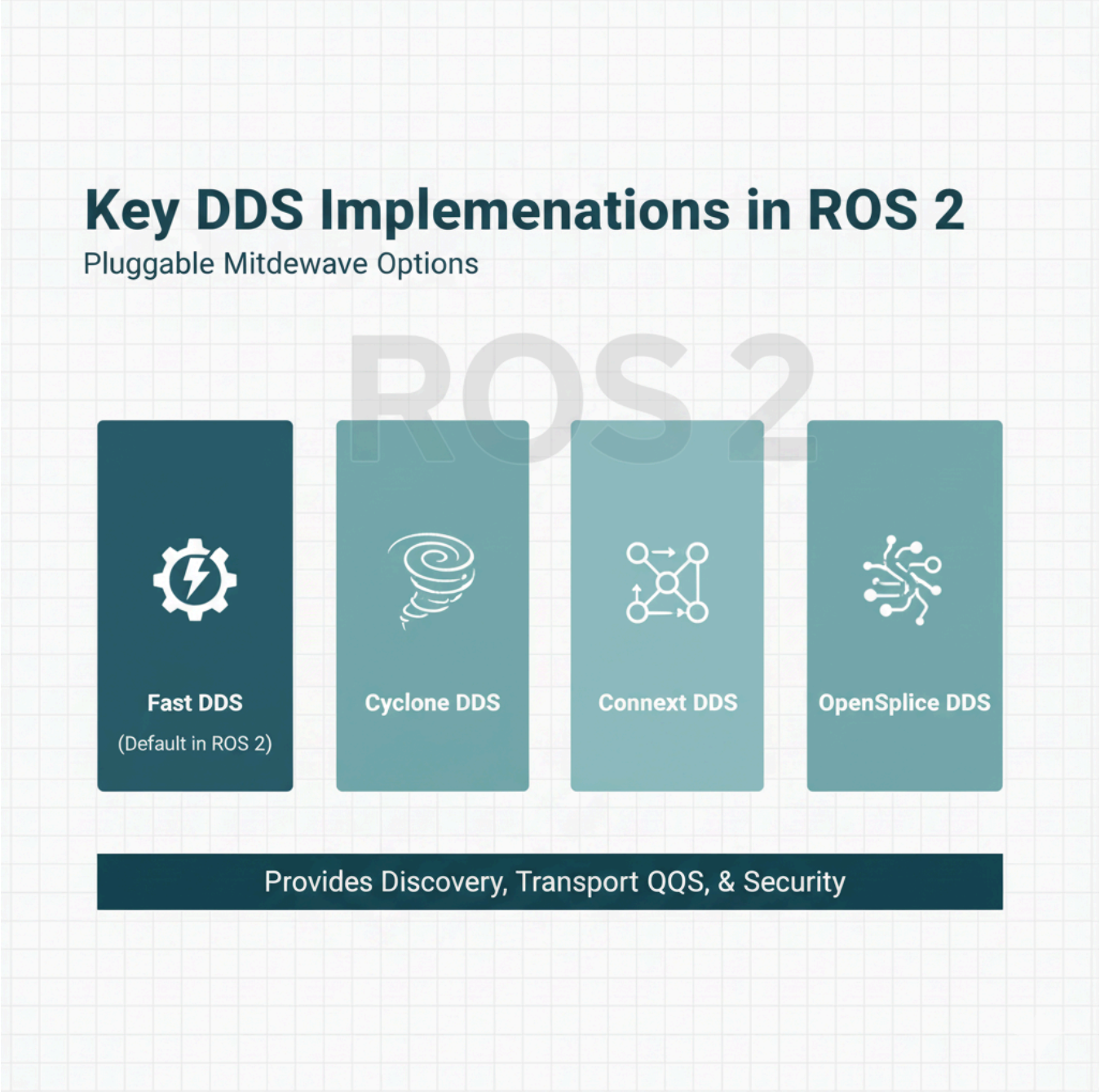
ROS 2 uses DDS (Data Distribution Service) as its middleware layer, which provides:

- Real-time data distribution
- Quality of Service (QoS) policies
- Discovery and transport mechanisms
- Platform independence

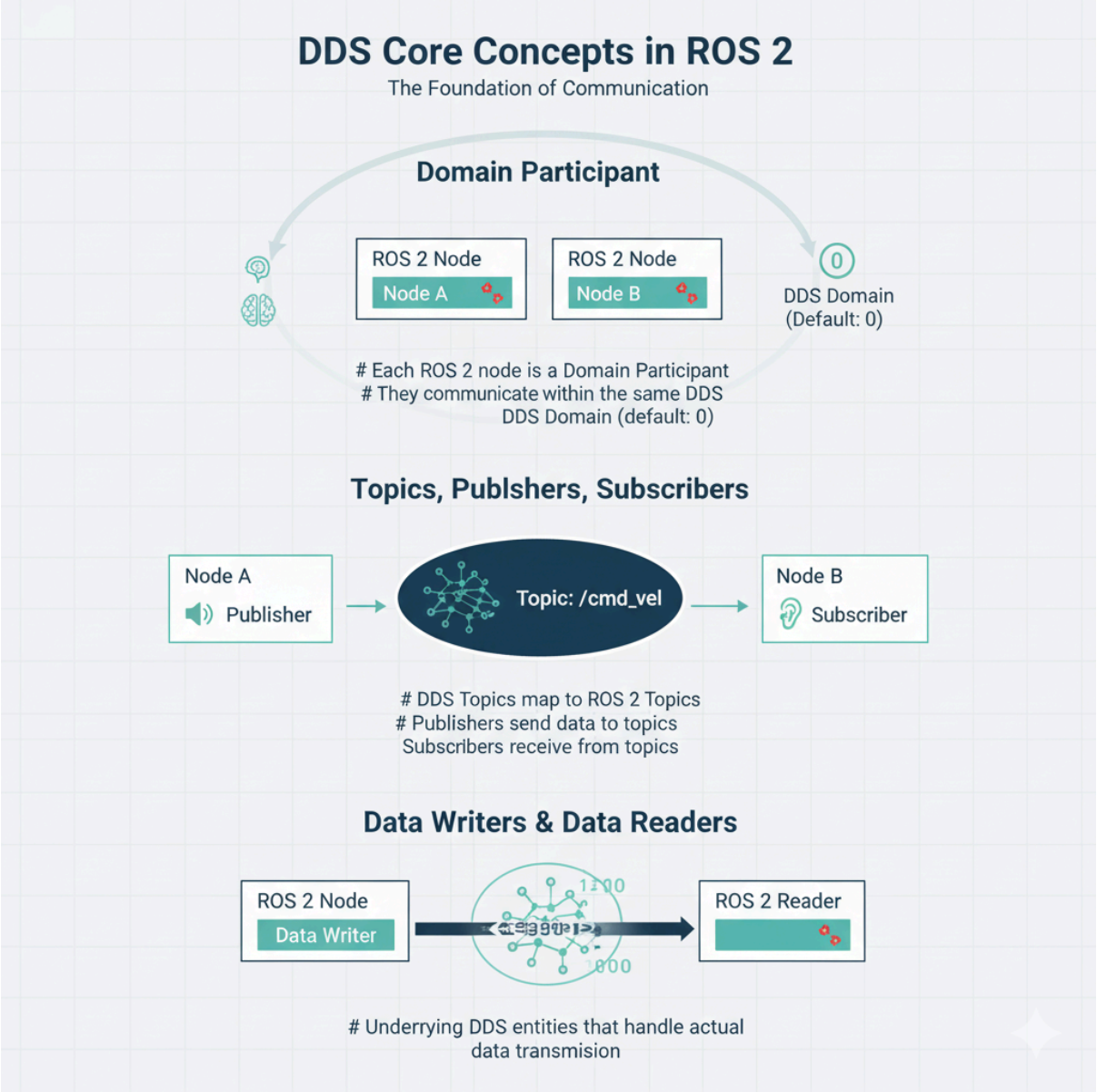


Key DDS Implementations in ROS 2

- Fast DDS (Default in ROS 2)
- Cyclone DDS
- Connext DDS
- OpenSplice DDS



2. DDS (Data Distribution Service) Concepts



DDS Discovery Process

1. Participant Discovery - Nodes find each other
2. Endpoint Discovery - Publishers/Subscribers match
3. Data Exchange - Actual message transmission

3. QoS (Quality of Service) Deep Dive

What is QoS?

QoS policies control how data is delivered, not just what data is delivered.

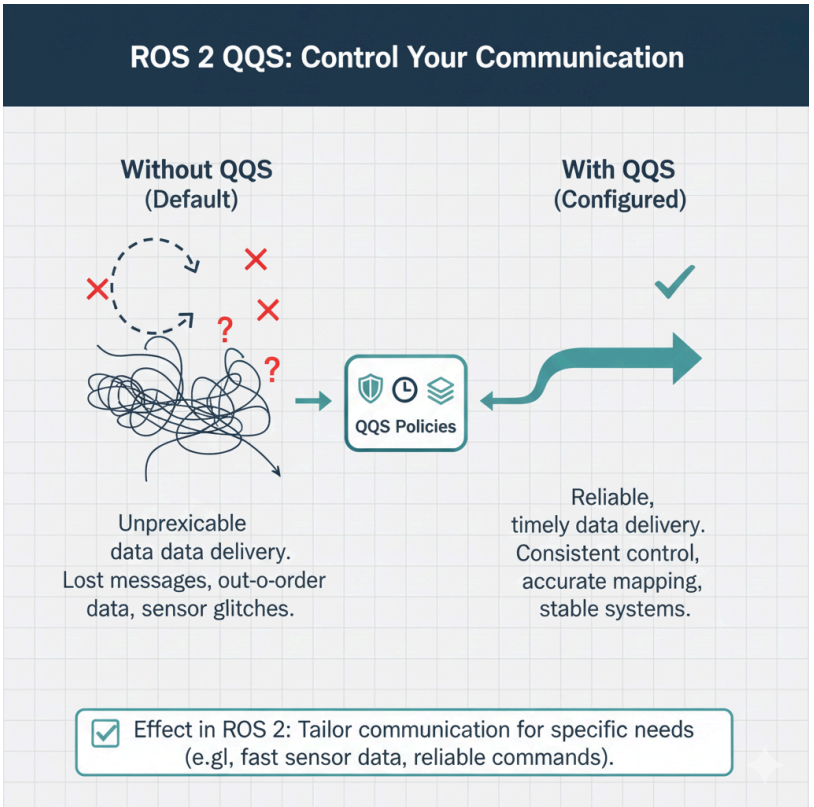
Key QoS Policies

QoS Policy	Purpose	Options
Reliability	Message delivery guarantee	`RELIABLE`, `BEST_EFFORT`
Durability	Message persistence	`VOLATILE`, `TRANSIENT_LOCAL`
History	Message queue management	`KEEP_LAST`, `KEEP_ALL`
Depth	Queue size for KEEP_LAST	Integer value
Deadline	Maximum time between messages	Duration
Lifespan	Maximum message validity	Duration
Liveliness	Node activity monitoring	`AUTOMATIC`, `MANUAL_BY_TOPIC`

QoS Profiles

ROS 2 provides predefined QoS profiles:

Profile	Use Case	Reliability	Durability	History
`SENSOR_DATA`	Sensor streams	`BEST_EFFORT`	`VOLATILE`	`KEEP_LAST`
`PARAMETERS`	Parameter updates	`RELIABLE`	`VOLATILE`	`KEEP_LAST`
`SERVICES_DEFAULT`	Services	`RELIABLE`	`VOLATILE`	`KEEP_LAST`
`SYSTEM_DEFAULT`	System topics	`RELIABLE`	`TRANSIENT_LOCAL`	`KEEP_LAST`



4. Practical QoS Examples

Let's create a comprehensive test package for QoS experiments.

You are building a communication system for a smart factory with different types of data streams. Each has different reliability and performance requirements.

📁 Project Structure

```
text
ros2_ws1/
└── src/
    └── factory_communication/
        ├── package.xml
        ├── setup.py
        ├── launch/
        │   └── factory_demo.launch.py
        └── factory_communication/
            ├── __init__.py
            ├── emergency_controller.py
            ├── sensor_monitor.py
            ├── production_line.py
            ├── calculator_service.py

            └── qos_analyzer.py
```

Scenario 1: Emergency Stop System (RELIABLE QoS)

Situation:

The factory has an emergency stop system that **must never lose messages**. A 100ms delay is acceptable, but message loss could cause safety hazards.

Code: `emergency_controller.py`

python

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from std_msgs.msg import String
import time

class EmergencyController(Node):
    def __init__(self):
        super().__init__('emergency_controller')

        # 🎯 CRITICAL: RELIABLE QoS for safety
        emergency_qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            durability=DurabilityPolicy.TRANSIENT_LOCAL,  # Persist for
late joiners
            depth=10
        )

        self.emergency_pub = self.create_publisher(
            String, '/factory/emergency', emergency_qos
        )

        # 🎯 Also subscribe to emergency stops
        self.emergency_sub = self.create_subscription(
            String, '/factory/emergency', self.emergency_callback,
emergency_qos
        )

        self.timer = self.create_timer(2.0, self.send_heartbeat)
        self.heartbeat_count = 0

        self.get_logger().info('🚨 Emergency Controller Started (RELIABLE
QoS)')

    def send_heartbeat(self):
        msg = String()
        msg.data = f'EMERGENCY_HEARTBEAT_{self.heartbeat_count}'
        self.emergency_pub.publish(msg)
        self.heartbeat_count += 1

        self.get_logger().info(f'🚨 Emergency heartbeat sent: {msg.data}')
```

```
    def emergency_callback(self, msg):
        self.get_logger().warning(f'🚨 EMERGENCY RECEIVED: {msg.data}')
        # In real system: trigger emergency stop procedures

def main():
    rclpy.init()
    node = EmergencyController()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Scenario 2: Sensor Data Stream (BEST_EFFORT QoS)

🎯 Situation:

Factory sensors (temperature, vibration) produce high-frequency data. Occasional loss is acceptable, but low latency is critical for real-time monitoring.

📝 Code: `sensor_monitor.py`

```
python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy
from std_msgs.msg import String
import random
import time

class SensorMonitor(Node):
    def __init__(self):
        super().__init__('sensor_monitor')

        # 🎯 HIGH-PERFORMANCE: BEST_EFFORT QoS for sensors
```

```
        sensor_qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            depth=5  # Small buffer for latest readings
        )

        # Multiple sensor publishers
        self.temp_pub = self.create_publisher(
            String, '/factory/sensors/temperature', sensor_qos
        )
        self.vibration_pub = self.create_publisher(
            String, '/factory/sensors/vibration', sensor_qos
        )

        # Subscribe to analyze data flow
        self.temp_sub = self.create_subscription(
            String, '/factory/sensors/temperature', self.temp_callback,
sensor_qos
        )

        self.timer = self.create_timer(0.1, self.publish_sensor_data)  # 10
Hz
        self.sensor_count = 0
        self.received_count = 0

        self.get_logger().info('📡 Sensor Monitor Started (BEST_EFFORT
QoS)')

    def publish_sensor_data(self):
        # 🎯 Simulate sensor data
        temp_msg = String()
        temp_msg.data = f'TEMP:{25 + random.uniform(-2, 2):.1f}C'
        self.temp_pub.publish(temp_msg)

        vibration_msg = String()
        vibration_msg.data = f'VIB:{random.uniform(0, 10):.2f}G'
        self.vibration_pub.publish(vibration_msg)

        self.sensor_count += 1

        if self.sensor_count % 50 == 0:
            loss_rate = ((self.sensor_count - self.received_count) /
self.sensor_count) * 100
            self.get_logger().info(
                f'📡 Sensors: Sent {self.sensor_count} | '
                f'Received {self.received_count} | '
                f'Loss: {loss_rate:.1f}%'
```

```
            )

    def temp_callback(self, msg):
        self.received_count += 1
        # Fast processing - no time for extensive logging
        if self.received_count % 20 == 0:
            self.get_logger().info(f'🌡️  Sensor data: {msg.data}')

def main():
    rclpy.init()
    node = SensorMonitor()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Scenario 3: Production Line Calculator Service

Situation:

The production line needs a calculator service for quality control calculations. Services inherently use RELIABLE QoS, but let's analyze the behavior.

Code: `calculator_service.py`

```
python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from example_interfaces.srv import AddTwoInts
import time

class CalculatorService(Node):
    def __init__(self):
        super().__init__('calculator_service')
```

```python
        # 🎯 SERVICES: Always use RELIABLE QoS internally
        self.service = self.create_service(
            AddTwoInts,
            '/factory/quality_calculator',
            self.calculate_callback
        )

        self.request_count = 0
        self.get_logger().info('🧮 Calculator Service Started')

    def calculate_callback(self, request, response):
        start_time = time.time()
        self.request_count += 1

        # 🎯 Simulate calculation for quality control
        response.sum = request.a + request.b

        processing_time = (time.time() - start_time) * 1000  # ms

        self.get_logger().info(
            f'🧮 Service Request {self.request_count}: '
            f'{request.a} + {request.b} = {response.sum} | '
            f'Processing: {processing_time:.1f}ms'
        )

        return response

def main():
    rclpy.init()
    node = CalculatorService()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.get_logger().info(f'🧮 Total service requests:
{node.request_count}')
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':

    main()
```

Scenario 4: Production Line Manager

Situation:

Coordinates multiple systems with mixed QoS requirements and uses the calculator service.

Code: `production_line.py`

```
python
```

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy
from std_msgs.msg import String
from example_interfaces.srv import AddTwoInts
import threading
import time

class ProductionLine(Node):
    def __init__(self):
        super().__init__('production_line')

        # 🎯 MIXED QoS SUBSCRIPTIONS
        reliable_qos = QoSProfile(reliability=ReliabilityPolicy.RELIABLE,
depth=10)
        best_effort_qos =
QoSProfile(reliability=ReliabilityPolicy.BEST_EFFORT, depth=5)

        # Subscribe to emergency (RELIABLE)
        self.emergency_sub = self.create_subscription(
            String, '/factory/emergency', self.emergency_callback,
reliable_qos
        )

        # Subscribe to sensors (BEST_EFFORT)
        self.sensor_sub = self.create_subscription(
            String, '/factory/sensors/temperature', self.sensor_callback,
best_effort_qos
        )

        # Service client for calculations
```

```
        self.calculator_client = self.create_client(AddTwoInts,
'/factory/quality_calculator')

        # Production status publisher
        self.status_pub = self.create_publisher(String,
'/factory/production_status', reliable_qos)

        self.timer = self.create_timer(5.0, self.production_cycle)
        self.cycle_count = 0

        self.get_logger().info('🏭 Production Line Started')

    def emergency_callback(self, msg):
        self.get_logger().error(f'🛑 PRODUCTION HALT: {msg.data}')
        # In real system: stop all machines safely

    def sensor_callback(self, msg):
        # Quick processing - don't block the thread
        if 'TEMP' in msg.data:
            temp = float(msg.data.split(':')[1].replace('C', ''))
            if temp > 27.0:
                self.get_logger().warning(f'🌡️  High temperature alert:
{temp}C')

    def call_calculator_service(self, a, b):
        if not self.calculator_client.wait_for_service(timeout_sec=2.0):
            self.get_logger().error('❌ Calculator service not available')
            return None

        request = AddTwoInts.Request()
        request.a = a
        request.b = b

        future = self.calculator_client.call_async(request)

        # 🎯 Simulate async service call
        def process_result(future):
            try:
                response = future.result()
                self.get_logger().info(f'✅ Quality calculation result:
{response.sum}')
            except Exception as e:
                self.get_logger().error(f'❌ Service call failed: {e}')

        future.add_done_callback(process_result)
        return future
```

```
    def production_cycle(self):
        self.cycle_count += 1

        status_msg = String()
        status_msg.data = f'PRODUCTION_CYCLE_{self.cycle_count}_RUNNING'
        self.status_pub.publish(status_msg)

        self.get_logger().info(f'🏭 Production cycle {self.cycle_count}
started')

        # 🎯 Use calculator service for quality control
        if self.cycle_count % 2 == 0:
            self.call_calculator_service(self.cycle_count, 10)

        if self.cycle_count >= 10:
            self.get_logger().info('🏭 Training demonstration complete')
            self.timer.cancel()

def main():
    rclpy.init()
    node = ProductionLine()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.get_logger().info(f'🏭 Completed {node.cycle_count} production
cycles')
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':

    main()
```

Scenario 5: QoS Performance Analyzer

Situation:

Monitor and analyze the performance of different QoS settings in real-time.

Code: `qos_analyzer.py`

```
python
```

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy
from std_msgs.msg import String
import time
import statistics

class QoSAnalyzer(Node):
    def __init__(self):
        super().__init__('qos_analyzer')

        # 🎯 MONITOR ALL TOPICS WITH DIFFERENT QoS
        reliable_qos = QoSProfile(reliability=ReliabilityPolicy.RELIABLE,
depth=10)
        best_effort_qos =
QoSProfile(reliability=ReliabilityPolicy.BEST_EFFORT, depth=5)

        # Monitor reliable topics
        self.emergency_sub = self.create_subscription(
            String, '/factory/emergency', self.monitor_reliable,
reliable_qos
        )
        self.status_sub = self.create_subscription(
            String, '/factory/production_status', self.monitor_reliable,
reliable_qos
        )

        # Monitor best effort topics
        self.temp_sub = self.create_subscription(
            String, '/factory/sensors/temperature',
self.monitor_best_effort, best_effort_qos
        )
```

```
        self.vibration_sub = self.create_subscription(
            String, '/factory/sensors/vibration', self.monitor_best_effort,
best_effort_qos
        )

        # Statistics
        self.reliable_messages = {'emergency': [], 'status': []}
        self.best_effort_messages = {'temperature': [], 'vibration': []}
        self.start_time = time.time()

        self.analysis_timer = self.create_timer(10.0,
self.analyze_performance)

        self.get_logger().info('📊 QoS Analyzer Started - Monitoring all
communications')

    def monitor_reliable(self, msg):
        topic = msg.data.split('_')[0] if '_' in msg.data else 'unknown'
        current_time = time.time()

        if 'EMERGENCY' in msg.data:
            self.reliable_messages['emergency'].append(current_time)
            self.get_logger().warning(f'🚨 Reliable message: {msg.data}')
        else:
            self.reliable_messages['status'].append(current_time)
            self.get_logger().info(f'📋 Status message: {msg.data}')

    def monitor_best_effort(self, msg):
        sensor_type = msg.data.split(':')[0]
        current_time = time.time()

        if sensor_type in self.best_effort_messages:
            self.best_effort_messages[sensor_type].append(current_time)

        # Minimal logging for performance
        if len(self.best_effort_messages[sensor_type]) % 30 == 0:
            self.get_logger().info(f'📡 {sensor_type} samples:
{len(self.best_effort_messages[sensor_type])}')

    def analyze_performance(self):
        current_time = time.time()
        elapsed = current_time - self.start_time

        self.get_logger().info('=' * 60)
        self.get_logger().info('📊 QoS PERFORMANCE ANALYSIS:')
```

```
        # 🎯 Analyze RELIABLE topics
        reliable_total = sum(len(msgs) for msgs in
self.reliable_messages.values())
        reliable_rate = reliable_total / elapsed if elapsed > 0 else 0
        self.get_logger().info(f'🔒 RELIABLE QoS: {reliable_total} messages
| Rate: {reliable_rate:.1f} msg/s')

        # 🎯 Analyze BEST_EFFORT topics
        best_effort_total = sum(len(msgs) for msgs in
self.best_effort_messages.values())
        best_effort_rate = best_effort_total / elapsed if elapsed > 0 else
0
        self.get_logger().info(f'⚡ BEST_EFFORT QoS: {best_effort_total}
messages | Rate: {best_effort_rate:.1f} msg/s')

        # 🎯 Calculate approximate loss rate (simplified)
        if best_effort_total > 0:
            # Assuming sensors publish at ~10Hz each (2 sensors × 10Hz ×
elapsed)
            expected_messages = 2 * 10 * elapsed
            if expected_messages > 0:
                delivery_rate = (best_effort_total / expected_messages) *
100
                self.get_logger().info(f'📈 BEST_EFFORT delivery rate:
~{delivery_rate:.1f}%')

        self.get_logger().info('=' * 60)

def main():
    rclpy.init()
    node = QoSAnalyzer()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        # Final analysis
        node.analyze_performance()
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Complete Launch File

`launch/factory_demo.launch.py`

python

```python
#!/usr/bin/env python3
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        # 🚨 Emergency System (RELIABLE QoS)
        Node(
            package='factory_communication',
            executable='emergency_controller',
            name='emergency_controller',
            output='screen'
        ),

        # 📡 Sensor System (BEST_EFFORT QoS)
        Node(
            package='factory_communication',
            executable='sensor_monitor',
            name='sensor_monitor',
            output='screen'
        ),

        # 🧮 Calculator Service
        Node(
            package='factory_communication',
            executable='calculator_service',
            name='calculator_service',
            output='screen'
        ),

        # 🏭 Production Line (Mixed QoS)
        Node(
            package='factory_communication',
            executable='production_line',
            name='production_line',
            output='screen'
        ),
```

```
        # 📊 QoS Analyzer
        Node(
            package='factory_communication',
            executable='qos_analyzer',
            name='qos_analyzer',
            output='screen'
        ),

    ])
```

Run the Complete System

```
bash
cd ~/ros2_ws1

ros2 launch factory_communication factory_demo.launch.py
```

Monitor QoS Behavior

```
bash
# Terminal 2: Monitor topics
ros2 topic list
ros2 topic hz /factory/emergency
ros2 topic hz /factory/sensors/temperature

# Terminal 3: Check QoS settings
ros2 topic info /factory/emergency --verbose
ros2 topic info /factory/sensors/temperature --verbose

# Terminal 4: Service calls

ros2 service call /factory/quality_calculator
example_interfaces/srv/AddTwoInts "{a: 5, b: 3}"
```

Analyze Performance

Watch the analyzer output to see:

- Reliable QoS: Consistent but slower message rates
- Best Effort QoS: Higher rates but potential message loss
- Service calls: Reliable request-response pattern

Test Under Load

```bash
# Create network stress
ros2 topic pub /stress_test std_msgs/String "data: 'test'" --rate 100

# Observe how different QoS profiles handle the load
```

QoS Policies in Our Factory:

Component	QoS Profile	Reliability	Use Case
Emergency System	`RELIABLE`	✅ Guaranteed	Safety commands
Sensor Network	`BEST_EFFORT`	⚡ Maximum speed	Sensor data
Calculator Service	`SERVICE_DEFAULT`	🔄 Request-Response	Quality control

Key QoS Policies:

```python
# RELIABLE QoS (for emergency system)
reliable_qos = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,       # 🎯 Guaranteed delivery
    durability=DurabilityPolicy.TRANSIENT_LOCAL, # 🎯 Persist for late joiners
    depth=10                                      # 🎯 Queue size
)
```

```python
# BEST_EFFORT QoS (for sensors)
best_effort_qos = QoSProfile(
    reliability=ReliabilityPolicy.BEST_EFFORT,    # 🎯 Maximum performance
    durability=DurabilityPolicy.VOLATILE,         # 🎯 No persistence
    depth=5                                       # 🎯 Small buffer
)
```