ADMIN:DOCS

Neuro-symbolic AI: understanding and applying the best of both worlds

Neuro-symbolic AI combines neural models, knowledge graphs and rule-based logic. Hybrid systems use that mix for explainable RAG, checkable answers and production-grade AI workflows.

Category: Neuro-Symbolic AI **Published:** 25.11.2024 **Last reviewed:** 09.09.2026

Neural models recognise patterns, draft answers and find similarities in huge data spaces. They can also hallucinate, miss rules or make a technically wrong answer sound convincing. Symbolic systems work differently: they know explicit facts, relations, rules and check paths, but they do not learn on their own from raw data.

Neuro-symbolic AI joins those two ways of working. The approach matters wherever LLMs work with enterprise knowledge, knowledge graphs, GraphRAG or rule-based checks, and an answer has to stay traceable.

Why neuro-symbolic AI?

The problem with purely neural approaches

Neural networks are everywhere today — from speech recognition to image processing. They still have a basic weakness:

Typical weaknesses:

- Decisions are often hard to explain.
- Training data demand is high.
- Edge cases can be handled unstably.
- Hard logic, policies or domain exclusions are missing without an extra layer.

Source: https://admindocs.de/en/neuro-symbolic-ai/neuro-symbolic-ai-understanding-and-applying-the-best-of-both-worlds

The issue is not pattern recognition itself, but the missing justification. A model can classify an X-ray, a logfile or a user question correctly and still fail to explain which rule, source or domain constraint carries the decision. In critical applications such as medical diagnostics or autonomous driving, that is not enough.

The problem with purely symbolic approaches

Symbolic AI systems work with explicit rules and logic. They are interpretable, but rigid:

Typical weaknesses:

- The system does not learn on its own from new examples.
- Noisy or incomplete inputs make rules brittle quickly.
- Pattern recognition has to be modelled explicitly in advance.
- Every new rule raises maintenance and test cost.

A symbolic system works well as long as facts and rules are modelled completely enough. As soon as inputs are noisy, new patterns appear or terms do not match the knowledge base exactly, it becomes brittle.

The solution: a hybrid approach

Combining both approaches promises:

The hybrid approach splits the work cleanly:

- The neural model recognises patterns in images, texts, logs or measurements.
- The symbolic layer checks facts, rules, dependencies and exceptions.
- The knowledge base holds terms, relations and domain constraints.
- The result stays more explainable, because you get a check path, not only a score.

For LLM applications that means in practice: the model generates or analyses language, a knowledge graph supplies structured relations, rules set limits, and a validation step checks the answer against reliable knowledge.

Source: https://admindocs.de/en/neuro-symbolic-ai/neuro-symbolic-ai-understanding-and-applying-the-best-of-both-worlds

Fundamentals: the two worlds

Neural networks: learning patterns from data

Neural networks consist of layers of artificial neurons that process input data and recognise patterns:

```
import numpy as np

class SimpleNeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialise weights randomly
        self.weights_input_hidden = np.random.randn(input_size, hidden_size) * 0.1
        self.weights_hidden_output = np.random.randn(hidden_size, output_size) * 0.1

    def sigmoid(self, x):
        """Activation function – makes the network non-linear"""
        return 1 / (1 + np.exp(-x))

    def forward(self, x):
        """Forward pass – from input to output"""
        # Layer 1: input → hidden layer
        hidden = self.sigmoid(np.dot(x, self.weights_input_hidden))

        # Layer 2: hidden layer → output
        output = self.sigmoid(np.dot(hidden, self.weights_hidden_output))

        return output

# Simple example: the network learns AND logic
nn = SimpleNeuralNetwork(2, 4, 1)
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [0], [0], [1]])  # AND logic

# Training
for epoch in range(10000):
    output = nn.forward(X)
    # Backpropagation would run here
```

```
        error = y - output

    print("Predictions after training:")
    for i in range(len(X)):
        prediction = nn.forward(X[i])
        print(f"  {X[i]} → {prediction[0]:.3f} (expected: {y[i][0]})")
```

The network recognises patterns in the data, but it cannot explain *why* it takes a given decision.

Symbolic AI: logic and rules

Symbolic systems work with formal rules and logic:

```
class SymbolicRuleEngine:
    def __init__(self):
        self.rules = []
        self.knowledge_base = {}

    def add_rule(self, condition, conclusion):
        """Add a rule: if condition, then conclusion"""
        self.rules.append({
            'condition': condition,
            'conclusion': conclusion
        })

    def add_fact(self, key, value):
        """Add a fact to the knowledge base"""
        self.knowledge_base[key] = value

    def evaluate(self):
        """Evaluate all rules"""
        results = []
        for rule in self.rules:
            if self.check_condition(rule['condition']):
                results.append(rule['conclusion'])
        return results
```

```
        def check_condition(self, condition):
            """Check whether a condition is met"""
            for key, value in condition.items():
                if self.knowledge_base.get(key) != value:
                    return False
            return True

    # Example: pet rule system
    engine = SymbolicRuleEngine()

    # Facts
    engine.add_fact('has_fur', True)
    engine.add_fact('goes_meow', True)

    # Rules
    engine.add_rule(
        {'has_fur': True, 'goes_meow': True},
        "→ It is a cat"
    )
    engine.add_rule(
        {'has_fur': True, 'goes_woof': True},
        "→ It is a dog"
    )

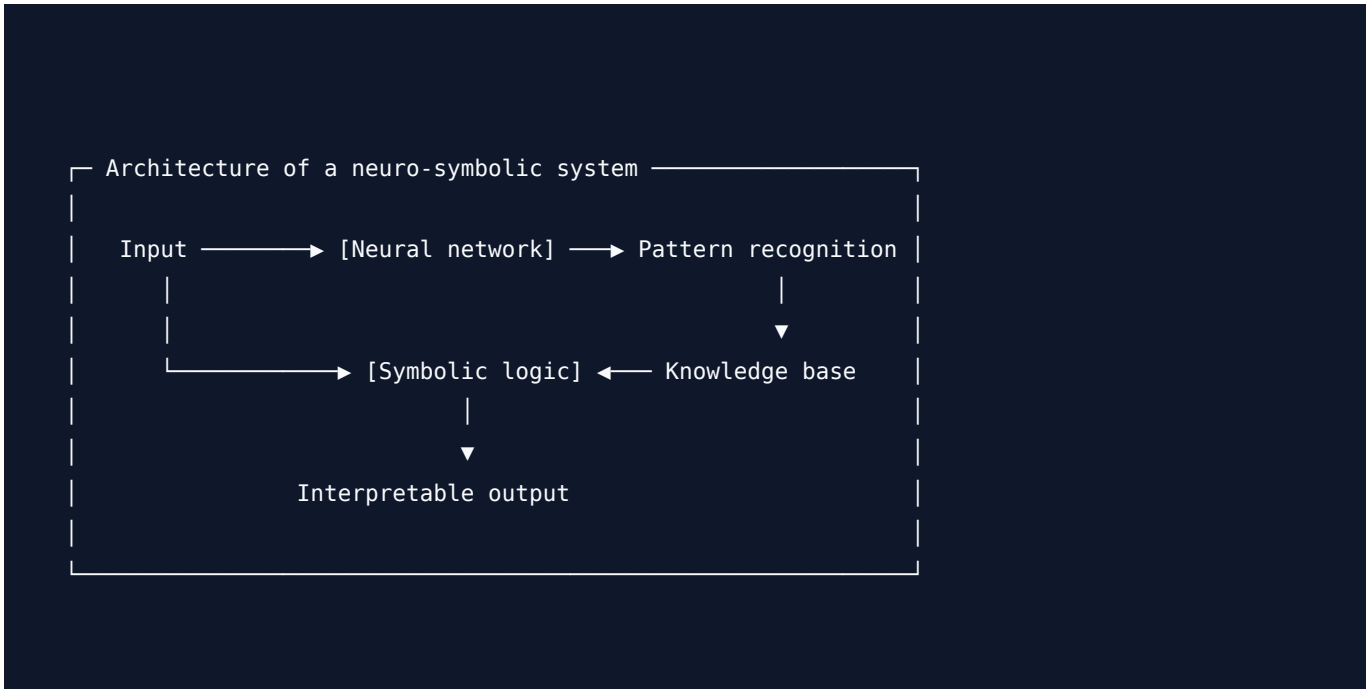
    # Evaluation
    results = engine.evaluate()
    for result in results:
        print(result)  # Output: → It is a cat
```

The system can justify its decisions logically, but it cannot recognise patterns in data that are not defined explicitly as rules.

The combination: how neuro-symbolic AI works

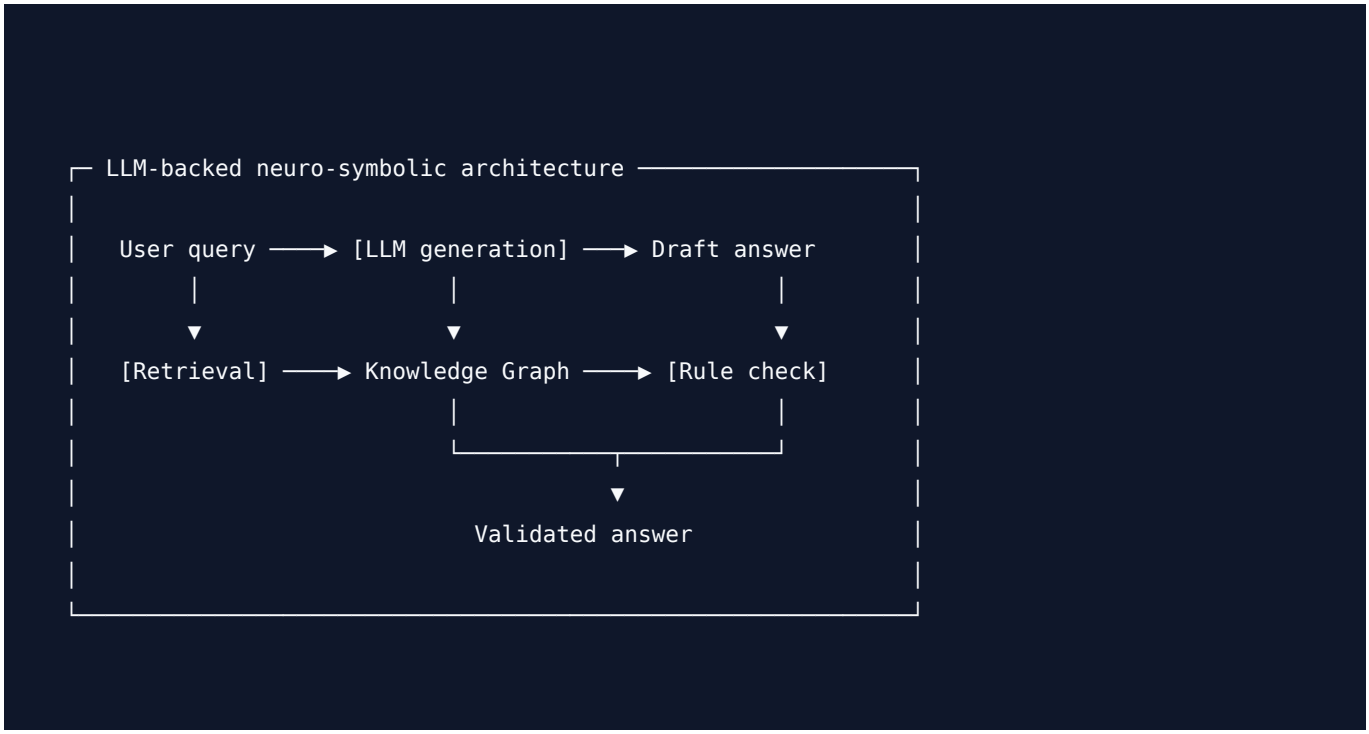
Architecture of a hybrid system

A typical neuro-symbolic system combines several components:



Modern variant: LLM, knowledge graph and RAG

In many systems a single model does not replace the whole logic. An LLM works with retrieval, a knowledge graph and explicit check rules instead. The knowledge graph makes entities and relations visible, retrieval supplies matching sources, and the LLM turns that into an answer. The symbolic layer then checks whether the answer fits known facts, permissions, policies or process rules.



Source: https://admindocs.de/en/neuro-symbolic-ai/neuro-symbolic-ai-understanding-and-applying-the-best-of-both-worlds

A plain vector RAG stack is not a neuro-symbolic system yet. The symbolic share appears only when relations, rules, taxonomies, ontologies or checkable paths take an active role in the answer logic. GraphRAG and KG-RAG are typical variants, because they do not only search similar passages. They use structured connections across documents, people, systems, risks or processes.

Practical example: pet recognition

A practical example is a system that recognises pets from images and can also explain why it took that decision.

```
class HybridPetRecognizer:
    def __init__(self):
        # Neural network for image processing
        self.neural_net = self._create_neural_net()

        # Symbolic rule set for explanations
        self.rule_engine = SymbolicRuleEngine()
        self._setup_rules()

    def _create_neural_net(self):
        """Create a simple neural network"""
        # In practice: a TensorFlow/PyTorch model
        return SimpleNeuralNetwork(
            input_size=64*64*3,  # RGB image
            hidden_size=128,
            output_size=3  # cat, dog, bird
        )

    def _setup_rules(self):
        """Set up symbolic rules"""
        self.rule_engine.add_rule(
            {'detection': 'cat', 'fur': True},
            "→ Cat recognised: carnivore with claws"
        )
        self.rule_engine.add_rule(
            {'detection': 'dog', 'fur': True},
```

```
                "→ Dog recognised: loyal companion"
            )

        def predict_with_explanation(self, image):
            """Prediction with explanation"""
            # 1. Neural network: pattern recognition
            prediction = self.neural_net.forward(image.flatten())
            pet_class = np.argmax(prediction)
            confidence = prediction[pet_class]

            # 2. Symbolic logic: explanation
            self.rule_engine.add_fact('detection',
                ['cat', 'dog', 'bird'][pet_class])
            self.rule_engine.add_fact('fur', True)  # example

            explanations = self.rule_engine.evaluate()

            return {
                'pet': ['Cat', 'Dog', 'Bird'][pet_class],
                'confidence': f"{confidence*100:.1f}%",
                'explanation': explanations[0] if explanations else "No explanation"
            }

    # Usage
    recognizer = HybridPetRecognizer()

    # Simulate an image (in practice: load a real image)
    fake_image = np.random.rand(64, 64, 3)

    result = recognizer.predict_with_explanation(fake_image)
    print(f"Result: {result['pet']}")
    print(f"Confidence: {result['confidence']}")
    print(f"Explanation: {result['explanation']}")
```

The system can now not only recognise *what* is in an image, but also explain *why* it took that decision.

Source: https://admindocs.de/en/neuro-symbolic-ai/neuro-symbolic-ai-understanding-and-applying-the-best-of-both-worlds

Application areas

Medical diagnostics

In medicine, explainability matters especially — doctors need to understand why an AI suggests a given diagnosis:

Typical check path:

- The CNN recognises conspicuous patterns in the X-ray.
- Diagnosis rules match the detection against clinical guidelines.
- Patient history and lab data constrain the interpretation.
- The recommendation contains not only a result, but also the clinical rationale.

A purely neural network might detect a tumour on an X-ray, but a hybrid system can also say: "The tumour is likely malignant because it has an irregular shape (neural recognition) and because clinical guidelines describe a matching growth pattern (symbolic logic)."

Autonomous driving

In autonomous driving, decisions have to be immediate and explainable:

```
class AutonomousDrivingHybrid:
    def __init__(self):
        self.perception = NeuralPerception()  # Detects objects
        self.planner = SymbolicPlanner()       # Plans routes

    def decide(self, sensor_data):
        # 1. Neural perception
        objects = self.perception.detect(sensor_data)

        # 2. Symbolic planning
        self.planner.update_facts({
            'vehicles_present': len(objects['cars']) > 0,
```

```
        'pedestrians_detected': len(objects['pedestrians']) > 0,
        'speed_too_high': self.check_speed(objects)
    })

    # 3. Decision with explanation
    decision = self.planner.decide()

    return {
        'action': decision['action'],
        'reason': decision['explanation'],
        'safety': decision['confidence']
    }
```

Language processing

In language processing, a hybrid system can understand context and apply grammatical rules:

Typical check path:

- BERT or an LLM recognises the semantic meaning of the input.
- Grammar rules check sentence structure, parts of speech and allowed patterns.
- Named entities are matched against known terms or domain models.
- The output is corrected or rejected with a reason.

Enterprise knowledge systems and GraphRAG

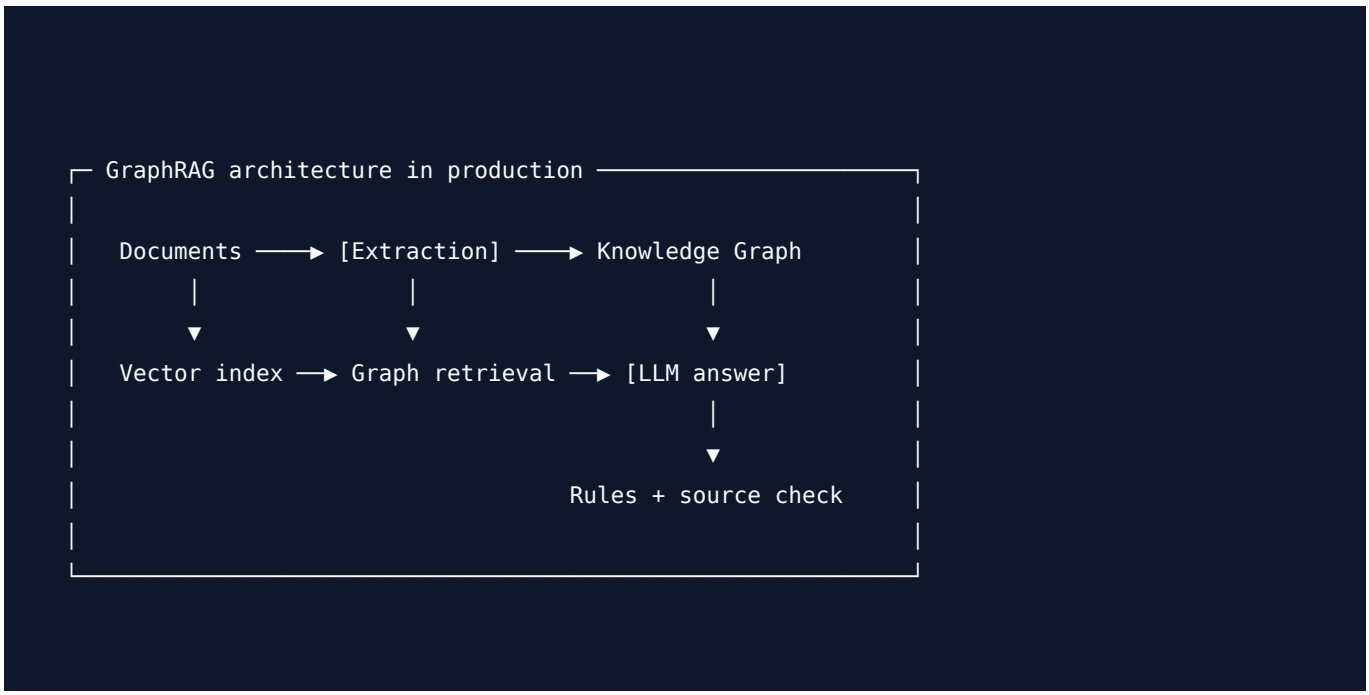
In companies, knowledge rarely sits cleanly in one database. It is spread across runbooks, tickets, contracts, product docs, policies and chat history. A neuro-symbolic approach puts that material into a form an assistant can check, not only phrase:

- The LLM extracts entities, relations and summaries from unstructured documents.
- The knowledge graph records which systems, teams, terms and dependencies belong together.
- Retrieval finds relevant evidence not only by text similarity, but also by graph neighbourhoods.
- Rules block answers that violate roles, compliance requirements or operational limits.

Source: https://admindocs.de/en/neuro-symbolic-ai/neuro-symbolic-ai-understanding-and-applying-the-best-of-both-worlds

- The answer can explain its path: source, relation, rule, conclusion.

GraphRAG in production



Error handling and challenges

Typical sources of error

Hybrid systems have their own failure modes: the neural model can be plausibly wrong, the rule set can be too tight, and the knowledge graph can hold stale or wrongly linked facts.

Typical sources of error:

- The neural model and the rule set reach different results.
- Training data or inputs are incomplete, biased or badly normalised.
- The knowledge base contains gaps, conflicting rules or stale facts.

1. Conflicts between components

Sometimes the neural network reaches a different result than the symbolic system. That has to be resolved:

```
class ConflictResolver:
    def __init__(self, neural_weight=0.6, symbolic_weight=0.4):
        self.neural_weight = neural_weight
        self.symbolic_weight = symbolic_weight

    def resolve(self, neural_result, symbolic_result):
        """Resolve conflicts between the systems"""
        if neural_result == symbolic_result:
            return neural_result, "Unified decision"

        # Weighted vote
        confidence_neural = self.get_neural_confidence()
        confidence_symbolic = self.get_symbolic_confidence()

        if confidence_neural * self.neural_weight > \
           confidence_symbolic * self.symbolic_weight:
            return neural_result, \
                f"Neural network prevails ({confidence_neural:.2f})"
        else:
            return symbolic_result, \
                f"Symbolic system prevails ({confidence_symbolic:.2f})"

    def get_neural_confidence(self):
        """Simulate neural confidence"""
        return 0.85  # example

    def get_symbolic_confidence(self):
        """Simulate symbolic confidence"""
        return 0.75  # example
```

2. Inconsistencies in the knowledge base

The symbolic system must stay current and consistent. Inconsistencies lead to wrong conclusions:

Source: https://admindocs.de/en/neuro-symbolic-ai/neuro-symbolic-ai-understanding-and-applying-the-best-of-both-worlds

```
class KnowledgeBaseValidator:
    def __init__(self):
        self.inconsistencies = []

    def validate(self, rules):
        """Check rules for inconsistencies"""
        for i, rule_a in enumerate(rules):
            for rule_b in rules[i+1:]:
                if self.contradicts(rule_a, rule_b):
                    self.inconsistencies.append(
                        (rule_a, rule_b)
                    )

        return len(self.inconsistencies) == 0

    def contradicts(self, rule_a, rule_b):
        """Check whether two rules contradict each other"""
        # Simplified check
        return (rule_a['condition'] == rule_b['condition'] and
                rule_a['conclusion'] != rule_b['conclusion'])
```

3. Data quality for the neural network

The neural network is only as good as its training data. Here is an approach to improving data:

```
class DataPreprocessor:
    def __init__(self):
        self.quality_metrics = {}

    def assess_quality(self, dataset):
        """Assess data quality"""
        metrics = {
            'completeness': self.check_completeness(dataset),
```

```
            'consistency': self.check_consistency(dataset),
            'representativeness': self.check_representation(dataset)
        }

        self.quality_metrics = metrics
        return metrics

    def check_completeness(self, dataset):
        """Check for missing values"""
        missing = sum(1 for row in dataset if any(
            v is None for v in row.values()
        ))
        return 1 - (missing / len(dataset))

    def improve_dataset(self, dataset):
        """Improve the dataset"""
        cleaned = []
        for row in dataset:
            if self.is_quality_acceptable(row):
                cleaned.append(self.normalize(row))
        return cleaned
```

LLM- and GraphRAG-specific risks

In LLM-based architectures the failure modes shift. The model can phrase an answer convincingly even though the graph only supplies weak hints. A graph can merge an entity wrongly, a policy can be stale, or retrieval can find the right documents but promote the wrong passage.

Typical checks:

- Do the extracted entities match unique IDs in the knowledge graph?
- Is the source current enough for the decision?
- Was the answer checked against hard rules, rather than only scored by the LLM?
- Can the system output the graph path or rule chain it used?
- Is there a fallback when graph, retrieval and model disagree?

Source: https://admindocs.de/en/neuro-symbolic-ai/neuro-symbolic-ai-understanding-and-applying-the-best-of-both-worlds

> ⚠ **Operational issue:** An LLM can still hallucinate despite retrieval. Retrieval supplies context, not a guarantee of truth. The symbolic layer therefore has to run concrete checks: fact matching, rule validation, source age, permissions and conflict resolution.

Integration problems

Integrating both systems is often the largest challenge:

Typical integration problems:

- The neural model returns a format the rule set does not expect.
- Confidence values, class labels or entities are interpreted differently.
- Rules contradict model assumptions or intervene too early in the flow.

Solution through systematic checks:

```
class SystemCheck:
    def __init__(self):
        self.checks = []

    def run_diagnostics(self):
        """Run a systematic system check"""
        print("Starting system diagnostics...")

        # 1. Check image processing
        print("Checking image processing...")
        if self.test_image_processing():
            print("✓ Image processing OK")

        # 2. Check the rule set
        print("Checking rule engine...")
        if self.test_rule_engine():
            print("✓ Rule engine OK")
```

```
        # 3. Check integration
        print("Checking the full system...")
        if self.test_integration():
            print("✓ Integration OK")
```

Best practices and optimisations

A neuro-symbolic system becomes production-ready only when knowledge, model behaviour, check paths and operations are maintained together. Otherwise you only get a more complex stack with unclear ownership.

Important optimisation areas:

- Performance: measure expensive model calls, graph traversals and validations.
- Reliability: test conflict cases, empty retrieval results and stale facts.
- Maintainability: keep rules, graph schema and model adapters separate.

1. Define the knowledge model and its limits

The most important architecture decision sits before the first model call: which facts, relations and rules belong in the system explicitly? For current LLM stacks that means: the knowledge graph needs a clear schema, retrieval needs quality limits, and the answer logic needs rules for uncertainty.

Good starting questions:

- Which entities are unique in the system: people, systems, services, documents, tickets, risks?
- Which relations matter for decisions: belongs to, depends on, replaces, contradicts, approves?
- Which rules are hard and must not be overridden by the model?
- Which answers must show sources, a graph path or a rule rationale?

- When must the system stop and ask for human review?

2. Code structure

> **Why does structure matter?**
>
> - Easier to understand
> - Easier to maintain
> - Easier to extend

Example of a clean structure:

```
# config.py – central configuration
class Config:
    """
    All important settings in one place
    Why? -> easier to change and manage
    """
    # Image processing
    IMAGE_SIZE = (64, 64)
    COLOR_CHANNELS = 3

    # Timing
    FEEDING_TOLERANCE = 15  # minutes

    # System settings
    DEBUG_MODE = True
    LOG_LEVEL = 'INFO'

# Usage in the main programme
from config import Config

def process_image(image):
    """Process images to the configured standard"""
```

```
    resized = cv2.resize(image, Config.IMAGE_SIZE)
    return resized
```

3. Logging and monitoring

> **Why does logging matter?**
>
> - You see what your system is doing
> - You find faults faster
> - You understand system behaviour better

```
# logger.py
import logging

def setup_logging():
    """Set up a clear logging system"""
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler('system.log'),
            logging.StreamHandler()  # also print to the console
        ]
    )

# Example of useful logging
def process_pet_image(image):
    """Process a pet image with detailed logging"""
    logging.info("Starting image processing")

    try:
        result = detector.predict(image)
        logging.info(f"Recognition succeeded: {result}")
```

```
        except Exception as e:
            logging.error(f"Recognition failed: {str(e)}")
            raise
```

4. Performance optimisation

Systematic performance improvement:

- Analyse: find slow model calls, graph queries and conversions.
- Optimise: touch only the measured bottlenecks.
- Validate: check that answer quality and explainability still hold.

Example of a performance improvement:

```
# Before optimisation
def process_images(image_list):
    """Process a list of images – too slow for many images"""
    results = []
    for image in image_list:
        result = process_single_image(image)
        results.append(result)
    return results

# After optimisation
from concurrent.futures import ThreadPoolExecutor

def process_images_optimized(image_list):
    """Process images in parallel – faster through concurrent work"""
    with ThreadPoolExecutor(max_workers=4) as executor:
        results = list(executor.map(process_single_image, image_list))
    return results
```

Command Reference (Cheatsheet)

Building block	Purpose	Typical use
`SymbolicRuleEngine.add_rule()`	register explicit rules	hard domain logic, explanations, compliance limits
`SymbolicRuleEngine.add_fact()`	write facts into the knowledge base	detected classes, user context, graph entities
`KnowledgeBaseValidator.validate()`	find rule conflicts	contradictory policies, inconsistent ontologies
`ConflictResolver.resolve()`	reconcile neural and symbolic results	model says A, rule set says B
`DataPreprocessor.assess_quality()`	assess training and input data	completeness, consistency, representativeness
`SystemCheck.run_diagnostics()`	check the integration path	image model, rule set, retrieval, graph and output
`GraphRAG` / `KG-RAG`	extend retrieval over relations	enterprise knowledge, complex questions, explainable answer paths
`ThreadPoolExecutor`	parallelise independent work	batch inference, preprocessing, non-blocking check jobs

Further Resources

Neural-Symbolic Reasoning over Knowledge Graphs From Symbolic to Neural and Back Neurosymbolic Retrievers for Retrieval-augmented Generation Microsoft Research: GraphRAG Microsoft GraphRAG Documentation

Conclusion

Neuro-symbolic AI is a practical architecture principle for systems that must not only answer, but also justify and verify. Classic examples such as pet recognition or medical diagnostics

Source: https://admindocs.de/en/neuro-symbolic-ai/neuro-symbolic-ai-understanding-and-applying-the-best-of-both-worlds

remain useful, but the centre of gravity has widened: LLMs, knowledge graphs, GraphRAG and rule-based validation make the approach especially relevant for enterprise knowledge, technical assistants and regulated workflows.

The decisive point is the split of roles. The neural model recognises patterns and phrases language. The knowledge graph structures facts and relations. Rules define limits. Check paths make visible why an answer was produced and when it must be rejected.

As in the ChatGPT Playbook, the goal is controllable AI use. The model is one component. The architecture decides which sources count, which rules are hard, and when the system must not give a reliable answer.

Source: https://admindocs.de/en/neuro-symbolic-ai/neuro-symbolic-ai-understanding-and-applying-the-best-of-both-worlds