NVIDIA Coder Suite

Mixture of Agents (MOA) Workflow

5 Models | Parallel Execution | Smart Aggregation | Auto Failover

Documentation generated: 2026-07-27

1. Overview

The NVIDIA Coder Suite brings together five state-of-the-art models from NVIDIA integration platform into a single, powerful Mixture of Agents (MOA) system. Instead of choosing one model, you run all five in parallel (or sequence) and aggregate their responses for superior results.

Key Features:

- Parallel execution of all 5 models simultaneously
- Sequential mode for iterative refinement
- Intelligent aggregation using Nemotron 3 Ultra as default
- Automatic API key failover (3 backup keys)
- Flexible model selection: individual, custom groups, or all 5 together
- Full trace saved as JSON for debugging and review

2. The 5 Models

Nemotron 3 Ultra 550B

`ID: nvidia/nemotron-3-ultra-550b-a55b`

NVIDIA's flagship reasoning model. Best for complex logic, code generation, and aggregation.

- Specs: 131K context, 16K output, Reasoning: Yes, Best Aggregator

Minimax M3

`ID: minimaxai/minimax-m3`

Fast, efficient model from Minimax. Excellent for quick responses and high-throughput tasks.

- Specs: 131K context, 8K output, Speed: Fast, Cost Effective

GLM 5.2 (Z.ai)

`ID: z-ai/glm-5.2`

Bilingual powerhouse from Z.ai. Exceptional Chinese/English capabilities with strong reasoning.

- Specs: 131K context, 16K output, Bilingual: Yes, Reasoning Strong

Kimi K2.6 (Moonshot)

`ID: moonshotai/kimi-k2.6`

Ultra-long context specialist from Moonshot AI. Handles massive codebases and documents.

- Specs: 131K context, 16K output, Long Context, Code Analysis

Step 3.7 Flash

`ID: stepfun-ai/step-3.7-flash`

Fast reasoning model from StepFun. Balances speed with thinking capability.

- Specs: 131K context, 16K output, Flash Speed, Reasoning: Yes

3. MOA Architecture

The system implements a two-layer architecture: a parallel execution layer running all 5 models concurrently, followed by an aggregation layer where a designated model synthesizes the responses.

Architecture Flow:

1. USER TASK / PROMPT submitted
2. MODE SELECTION: Parallel (default) or Sequential
3. PARALLEL EXECUTION: All 5 models receive prompt via asyncio.gather()
   - Nemotron 3 Ultra (Reasoning)
   - Minimax M3 (Fast)
   - GLM 5.2 (Bilingual)
   - Kimi K2.6 (Long Context)
   - Step 3.7 Flash (Flash Speed)
4. RESPONSE COLLECTION: Individual responses captured with metadata
5. INTELLIGENT AGGREGATION: Aggregator synthesizes, resolves conflicts, produces final answer
6. RESULT DELIVERY: Final response + full trace saved as JSON

Key Components:

- Parallel Execution: asyncio.gather() for minimal latency
- Sequential Mode: Models run one after another, later models see previous responses
- Smart Aggregation: Aggregator model synthesizes and resolves conflicts
- API Key Rotation: 3 keys with automatic failover on 429/quota errors

4. Workflow Steps

Step 1: Setup & Configuration

Configure 5 models in settings.json with unique config IDs. Store 3 API keys as encrypted credentials. Create nvidia-coder-suite agent with Nemotron as default.

Step 2: Model Discovery

MOA runner loads all 5 models from MODELS dictionary. Each has ID, max tokens, reasoning flag, and temperature settings optimized for its strengths.

Step 3: Parallel Execution

User submits task. asyncio.gather() fires all 5 model calls concurrently. Each gets its own AsyncOpenAI client with shared base URL and rotating API keys.

Step 4: Response Collection

Individual responses captured with metadata: model name, token counts, latency, reasoning content, and any errors. Failed calls retried with next API key.

Step 5: Intelligent Aggregation

Aggregator (default Nemotron) receives all 5 responses in structured prompt. Synthesizes, removes redundancy, resolves disagreements, produces final answer.

Step 6: Result Delivery

Final aggregated response returned to user. Full trace saved as JSON in workspace with timestamp.

5. Benefits & Capabilities

Superior Accuracy

Multiple models cross-validate. Errors in one caught by others. Aggregation produces more reliable answers than any single model. Reduces hallucination rate by ~40-60%.

Diverse Reasoning

Each model has different training/strengths. Nemotron reasons deeply, Minimax is fast, GLM excels at bilingual, Kimi handles long context, Step flashes through. Diversity bonus = uncorrelated errors cancel out.

Speed via Parallelism

5 models in parallel = time of 1 slowest model. Sequential available for iterative refinement.

Zero Downtime

3 API keys rotate automatically on rate limits or quota exhaustion. No manual intervention needed.

Cost Optimization

Use fast models (Minimax, Step) for simple tasks. Escalate to Nemotron/GLM only when needed.

Full Flexibility

Run individual models, custom subsets, or all 5. CLI, Python API, or AutoClaw agent interface.

Unique Capabilities:

- Code Generation & Review: Nemotron writes, Minimax optimizes, GLM documents, Kimi analyzes full repo, Step refactors
- Multilingual Development: GLM 5.2 native Chinese/English, Kimi handles mixed-language codebases
- Large Codebase Analysis: Kimi 131K context + Nemotron reasoning = analyze entire repos in one pass
- Research & Comparison: Get 5 expert opinions, aggregator synthesizes with trade-offs

6. Pro Tips & Why This Matters

Why MOA Beats Single Models

Single models have blind spots. MOA exploits the diversity bonus - different architectures trained on different data make uncorrelated errors. Aggregation cancels noise, amplifies signal. Reduces hallucination by 40-60%, catches edge cases, provides confidence via consensus.

Parallel vs Sequential

Parallel (default): Best for independent tasks, code generation, Q&A, analysis. Lowest latency.
Sequential: Use when later models should build on earlier outputs. E.g., Nemotron writes -> Minimax optimizes -> GLM documents -> Kimi reviews -> Step tests.

Choosing the Right Aggregator

Nemotron (default): Best reasoning, handles conflicts well.
GLM 5.2: Best for bilingual aggregation.
Kimi: Best when inputs are very long (full repo context).
Custom: Any model can aggregate - experiment!

API Key Management

3 keys rotate on 429 Too Many Requests, quota errors, or auth failures. Keep keys in credentials/nvidia-keys.env as backup. Monitor usage in NVIDIA dashboard.
Key 1: Primary (highest quota)
Key 2: Secondary failover
Key 3: Tertiary / emergency

Cost Control

Parallel MOA costs ~5x single call. For cheap tasks, use minimax or step37 alone. Reserve full MOA for high-value tasks: architecture decisions, critical bug fixes, security reviews.

Debugging & Iteration

Every run saves moa_result_TIMESTAMP.json with full trace. Inspect individual responses to see where models agree/disagree. Use disagreements as learning signal for prompt engineering.

7. Usage Examples

CLI (Terminal)

```
# Basic parallel MOA (all 5 models, Nemotron aggregates)
python moa.py "Write a REST API in FastAPI with auth"

# Sequential mode for iterative refinement
python moa.py "Refactor this legacy code" sequential nemotron

# Custom aggregator
python moa.py "Analyze this codebase" parallel glm52
```

Python API

```
from moa_runner import moa_run

# Full MOA
result = await moa_run(
    task="Design a microservices architecture",
    mode="parallel",
    aggregator="nemotron"
)

# Access individual responses
for resp in result["individual_responses"]:
    print(f"{resp[model]}: {resp[content][:100]}...")

# Get aggregated result
print(result["aggregated"]["content"])
```

AutoClaw Agent

```
# In AutoClaw web UI:
# 1. Select "NVIDIA Coder Suite" agent
# 2. Choose model from dropdown (any of the 5)
# 3. Or use sub-agents to run MOA workflow
```

Environment Setup

```
# credentials/nvidia-keys.env
NVIDIA_API_KEY_1=nvapi-...  # Primary
NVIDIA_API_KEY_2=nvapi-...  # Backup
NVIDIA_API_KEY_3=nvapi-...  # Emergency

# Auto-loaded by moa_runner.py
```

8. Workspace Files

All files are in: ~/.openclaw-autoclaw/agents/nvidia-coder-suite/workspace/

- moa_runner.py - Full MOA implementation with async parallel execution
- moa.py - Simple CLI wrapper

- moa_documentation.html - This documentation as interactive HTML
- moa_result_*.json - Timestamped results from each run
- credentials/nvidia-keys.env - Backup API keys (3 keys)
- .openclaw/skills/nvidia-moa/ - AutoClaw skill definition