GlobalMOO Developer Guide

Multi-Objective Optimization for VERILINGUA × VERIX

Version: 1.0
Audience: Developers implementing the cognitive architecture
Prerequisites: Python 3.10+, familiarity with the Developer Integration Guide v1.1
GlobalMOO Access: API key from globalmoo.com

---

0) What GlobalMOO provides (and why it matters)

Traditional optimization finds a single "best" solution by combining objectives into a weighted sum. This approach has a fundamental problem: you must choose weights before knowing the trade-off structure. How much is one point of accuracy worth in tokens? Is calibration twice as important as speed, or three times? These arbitrary choices hide the real trade-offs.

GlobalMOO takes a different approach. Instead of collapsing objectives into a single number, it maps the **Pareto frontier**: the set of all solutions where improving any objective necessarily worsens at least one other. This reveals the trade-off structure itself, letting you make informed decisions about which trade-offs to accept.

What GlobalMOO provides:

Efficient multi-objective search. Traditional evolutionary algorithms (NSGA-II, MOEA/D) might need 100,000 evaluations to find good solutions. GlobalMOO achieves comparable results in under 100 iterations. This matters when each evaluation requires an expensive LLM call.

Inverse problem solving. Traditional optimization asks "given these inputs, what outputs do I get?" GlobalMOO can answer the reverse: "given these desired outputs, what inputs produce them?" You can specify "I want accuracy > 0.9 and tokens < 500" and get configurations that achieve this, or learn that the combination is impossible.

Impact factor analysis. After optimization, GlobalMOO quantifies how much each input variable affects each output. This reveals which parts of your framework actually matter for which objectives, replacing intuition with measurement.

Model-agnostic operation. GlobalMOO treats your system as a black box. It doesn't need gradients, doesn't assume continuity, and works with mixed discrete and continuous variables. This fits our configuration space perfectly.

What GlobalMOO does NOT provide:

Prompt optimization. GlobalMOO searches over configuration vectors, not prompt text. Use DSPy for prompt expression optimization within each configuration cluster.

Evaluation metrics. You must define and compute your own objective functions. GlobalMOO optimizes whatever you measure.

Automatic improvement. GlobalMOO finds optimal configurations within your search space. If your framework structure is wrong, optimization won't fix it. That's what Level 1 DSPy analysis addresses.

---

1) Core concepts

1.1 The forward model contract

GlobalMOO optimizes a **forward model**: a function that takes input variables and produces output variables.

```
f(x₁, x₂, ..., xₙ) → (y₁, y₂, ..., yₘ)
```

In our architecture:

- Inputs are configuration parameters (frame enables, VERIX strictness, activation style, etc.)
- Outputs are measured outcomes (accuracy, calibration, tokens, latency, etc.)

Your job is to implement this forward model reliably. GlobalMOO handles the optimization.

```python
```

```python
def forward_model(config_vector: list[int]) -> list[float]:
    """
    The function GlobalMOO optimizes.

    Args:
        config_vector: A list of integers representing configuration choices
                      [frame1_on, frame2_on, ..., verix_strictness, ...]

    Returns:
        outcome_vector: A list of floats representing measured outcomes
                       [accuracy, calibration, tokens, latency, ...]
    """
    # Decode vector into configuration
    config = decode_config(config_vector)

    # Build prompts from configuration
    prompts = build_prompts(config)

    # Execute on task corpus
    responses = execute_tasks(prompts, task_corpus)

    # Measure outcomes
    outcomes = measure_outcomes(responses, task_corpus)

    # Encode as vector
    return encode_outcomes(outcomes)
```

1.2 Pareto optimality

A solution is **Pareto optimal** if you cannot improve any objective without worsening at least one other. The set of all Pareto optimal solutions forms the **Pareto frontier**.

Consider two objectives: accuracy (maximize) and tokens (minimize).

```
Configuration A: accuracy=0.85, tokens=400
Configuration B: accuracy=0.90, tokens=600
Configuration C: accuracy=0.80, tokens=350
Configuration D: accuracy=0.88, tokens=500
```

Which are Pareto optimal?

Configuration A is dominated by D (D is better on accuracy, same or better on tokens). Configuration C has the fewest tokens but lowest accuracy—nothing dominates it. Configuration B has the highest accuracy but most tokens—nothing dominates it. Configuration D beats A on accuracy with acceptable tokens—nothing dominates it.

Pareto frontier: {B, C, D}

The frontier reveals the trade-off: moving from C to D buys 8 points of accuracy for 150 tokens. Moving from D to B buys 2 more points for 100 more tokens. You can now make an informed choice based on what you value.

1.3 Inverse optimization

Forward optimization: "What outcomes does this configuration produce?"
Inverse optimization: "What configurations produce these outcomes?"

GlobalMOO's inverse capability lets you specify targets:

```python
targets = {
    "accuracy": {"type": "greater_than_equal", "value": 0.85},
    "calibration": {"type": "less_than_equal", "value": 0.15},
    "tokens": {"type": "less_than_equal", "value": 500},
    "latency": {"type": "less_than_equal", "value": 2.0}
}
```

GlobalMOO returns configurations that satisfy these constraints, or reports that no such configuration exists within the search space. This is how you query the system: "Find me a configuration that meets these requirements."

1.4 Impact factors

After collecting enough evaluations, GlobalMOO can compute **impact factors**: how much each input variable influences each output variable.

```
Impact of turkish_evidential on calibration: 0.35
Impact of turkish_evidential on accuracy: 0.08
Impact of verix_strictness on format_compliance: 0.52
Impact of verix_strictness on tokens: 0.28
```

This tells you:

- Turkish evidential frame strongly affects calibration but barely affects accuracy

- VERIX strictness strongly affects format compliance and moderately increases token usage

Impact factors guide framework evolution. Low-impact variables are candidates for removal. High-impact variables deserve careful tuning.

---

2) GlobalMOO API structure

GlobalMOO uses a hierarchical structure: Models contain Projects contain Trials. This organization lets you run multiple optimization experiments with different settings.

2.1 Entity hierarchy

```
Model (defines dimensions)
  └── Project (defines bounds and types)
        └── Trial (defines objectives and runs optimization)
              └── Objective (defines targets and constraints)
```

Model: Defines how many input and output variables your problem has. Create one model per problem structure. If you add or remove configuration parameters, you need a new model.

Project: Defines the bounds, types, and categories for input variables. A model can have multiple projects with different search bounds.

Trial: A specific optimization run. Load your seed data here, then run the optimization loop.

Objective: Defines what you're optimizing for—targets, constraints, and objective types for each output variable.

2.2 Input variable types

GlobalMOO supports several input types. For our configuration space, we primarily use:

INTEGER: Discrete values within bounds. Used for:

- Boolean flags (0 or 1)
- Enum indices (0, 1, 2 for strictness levels)
- Small integers (1-4 for max_frames)

CONTINUOUS: Real values within bounds. We don't currently use this, but it's available for parameters like learning rates or thresholds.

CATEGORICAL: Unordered categories. Could be used if we had non-ordinal choices, but our enums are ordinal so we use INTEGER.

2.3 Objective types

Each output variable gets an objective type that defines what "better" means:

Type	Meaning	Example
`MAXIMIZE`	Higher is better	accuracy
`MINIMIZE`	Lower is better	tokens, latency
`GREATER_THAN`	Must exceed threshold	accuracy > 0.8
`GREATER_THAN_EQUAL`	Must meet or exceed	accuracy >= 0.8
`LESS_THAN`	Must be below threshold	tokens < 600
`LESS_THAN_EQUAL`	Must be at or below	latency <= 2.0
`EQUAL`	Must match exactly	(rarely used)
`PERCENT`	Within percentage of target	calibration within 10% of 0.15

For Pareto optimization, use `MAXIMIZE` and `MINIMIZE`. For inverse queries with specific targets, use the threshold types.

---

3) Implementation

3.1 Installation and setup

bash

```bash
# Install the GlobalMOO SDK
pip install globalmoo-sdk

# Set your API key
export GLOBALMOO_API_KEY="your-api-key-here"
```

The SDK requires Python 3.8+ and provides a client for the GlobalMOO API.

3.2 Configuration encoding

First, define how to convert between configuration objects and vectors that GlobalMOO can optimize.

python

```python
from dataclasses import dataclass
from enum import IntEnum
from typing import List

class VERIXStrictness(IntEnum):
    OPTIONAL = 0
    PARTIAL = 1
    FULL = 2

class ActivationStyle(IntEnum):
    IMPLICIT = 0
    EXPLICIT = 1
    CHECKLIST = 2

@dataclass
class FrameworkConfig:
    russian_aspect: bool
    turkish_evidential: bool
    arabic_morphology: bool
    german_composition: bool
    japanese_keigo: bool
    chinese_classifiers: bool
    guugu_spatial: bool
    max_frames_per_task: int  # 1-4
    verix_strictness: VERIXStrictness
    verix_confidence_required: bool
    verix_ground_chain_required: bool

@dataclass
class PromptConfig:
    activation_style: ActivationStyle
    activation_verbosity: int  # 0-2
    fewshot_count: int  # 0-3
    show_verix_legend: bool
    system_prompt_length: int  # 0-2

@dataclass
class FullConfig:
    framework: FrameworkConfig
    prompt: PromptConfig



class VectorCodec:
```

```
"""
Encodes and decodes configurations as integer vectors for GlobalMOO.

The vector layout must be stable across all optimization runs.
Document it clearly and version it if you change it.
"""

# Define bounds for each position
# Format: (min, max) for each variable
BOUNDS = [
    (0, 1),   # 0: russian_aspect
    (0, 1),   # 1: turkish_evidential
    (0, 1),   # 2: arabic_morphology
    (0, 1),   # 3: german_composition
    (0, 1),   # 4: japanese_keigo
    (0, 1),   # 5: chinese_classifiers
    (0, 1),   # 6: guugu_spatial
    (1, 4),   # 7: max_frames_per_task
    (0, 2),   # 8: verix_strictness
    (0, 1),   # 9: verix_confidence_required
    (0, 1),   # 10: verix_ground_chain_required
    (0, 2),   # 11: activation_style
    (0, 2),   # 12: activation_verbosity
    (0, 3),   # 13: fewshot_count
    (0, 1),   # 14: show_verix_legend
    (0, 2),   # 15: system_prompt_length
]

INPUT_COUNT = 16

@property
def minimums(self) -> List[int]:
    return [b[0] for b in self.BOUNDS]

@property
def maximums(self) -> List[int]:
    return [b[1] for b in self.BOUNDS]

def to_vector(self, config: FullConfig) -> List[int]:
    """Convert a configuration to an integer vector."""
    return [
        int(config.framework.russian_aspect),
        int(config.framework.turkish_evidential),
        int(config.framework.arabic_morphology),
            int(config.framework.german_composition),
            int(config.framework.japanese_keigo),
            int(config.framework.chinese_classifiers),
            int(config.framework.guugu_spatial),
            config.framework.max_frames_per_task,
            int(config.framework.verix_strictness),
            int(config.framework.verix_confidence_required),
            int(config.framework.verix_ground_chain_required),
            int(config.prompt.activation_style),
            config.prompt.activation_verbosity,
            config.prompt.fewshot_count,
            int(config.prompt.show_verix_legend),
            config.prompt.system_prompt_length,
        ]

    def from_vector(self, vector: List[int]) -> FullConfig:
        """Convert an integer vector back to a configuration."""
        return FullConfig(
            framework=FrameworkConfig(
                russian_aspect=bool(vector[0]),
                turkish_evidential=bool(vector[1]),
                arabic_morphology=bool(vector[2]),
                german_composition=bool(vector[3]),
                japanese_keigo=bool(vector[4]),
                chinese_classifiers=bool(vector[5]),
                guugu_spatial=bool(vector[6]),
                max_frames_per_task=vector[7],
                verix_strictness=VERIXStrictness(vector[8]),
                verix_confidence_required=bool(vector[9]),
                verix_ground_chain_required=bool(vector[10]),
            ),
            prompt=PromptConfig(
                activation_style=ActivationStyle(vector[11]),
                activation_verbosity=vector[12],
                fewshot_count=vector[13],
                show_verix_legend=bool(vector[14]),
                system_prompt_length=vector[15],
            )
        )

    def validate_vector(self, vector: List[int]) -> bool:
        """Check that a vector is within bounds."""
        if len(vector) != self.INPUT_COUNT:
            return False
    for i, val in enumerate(vector):
        if not (self.BOUNDS[i][0] <= val <= self.BOUNDS[i][1]):
            return False
    return True
```

3.3 Outcome encoding

Similarly, define how to encode outcomes as vectors.

python

```python
from dataclasses import dataclass
from typing import List

@dataclass
class Outcomes:
    """
    Measured outcomes from evaluating a configuration.

    All values should be numeric. Higher is better for accuracy-like metrics,
    lower is better for cost-like metrics. Document the direction clearly.
    """
    task_accuracy: float        # 0-1, higher is better
    math_accuracy: float        # 0-1, higher is better
    calibration_score: float    # 0-1, LOWER is better (Brier score)
    format_compliance: float    # 0-1, higher is better
    ground_chain_validity: float  # 0-1, higher is better
    focus_score: float          # 0-1, higher is better
    nuance_appropriateness: float  # 0-1, higher is better
    tokens_total: float         # count, lower is better
    latency_seconds: float      # seconds, lower is better
    edge_case_robustness: float  # 0-1, higher is better
    epistemic_consistency: float  # 0-1, higher is better


class OutcomeCodec:
    """
    Encodes outcomes as vectors for GlobalMOO.
    """

    OUTPUT_COUNT = 11

    # Names for each output (for reporting)
    OUTPUT_NAMES = [
        "task_accuracy",
        "math_accuracy",
        "calibration_score",
        "format_compliance",
        "ground_chain_validity",
        "focus_score",
        "nuance_appropriateness",
        "tokens_total",
        "latency_seconds",
        "edge_case_robustness",
        "epistemic_consistency",
    ]

    # Direction: True = higher is better, False = lower is better
    OUTPUT_DIRECTIONS = [
        True,   # task_accuracy
        True,   # math_accuracy
        False,  # calibration_score (Brier, lower is better)
        True,   # format_compliance
        True,   # ground_chain_validity
        True,   # focus_score
        True,   # nuance_appropriateness
        False,  # tokens_total
        False,  # latency_seconds
        True,   # edge_case_robustness
        True,   # epistemic_consistency
    ]

    def to_vector(self, outcomes: Outcomes) -> List[float]:
        """Convert outcomes to a float vector."""
        return [
            outcomes.task_accuracy,
            outcomes.math_accuracy,
            outcomes.calibration_score,
            outcomes.format_compliance,
            outcomes.ground_chain_validity,
            outcomes.focus_score,
            outcomes.nuance_appropriateness,
            outcomes.tokens_total,
            outcomes.latency_seconds,
            outcomes.edge_case_robustness,
            outcomes.epistemic_consistency,
        ]

    def from_vector(self, vector: List[float]) -> Outcomes:
        """Convert a float vector back to outcomes."""
        return Outcomes(
            task_accuracy=vector[0],
            math_accuracy=vector[1],
            calibration_score=vector[2],
            format_compliance=vector[3],
            ground_chain_validity=vector[4],
            focus_score=vector[5],
            nuance_appropriateness=vector[6],
            tokens_total=vector[7],
            latency_seconds=vector[8],
            edge_case_robustness=vector[9],
            epistemic_consistency=vector[10],
        )
```

3.4 The GlobalMOO client wrapper

Now wrap the GlobalMOO SDK in a class that handles the optimization workflow.

python

```python
import os
import logging
from typing import List, Dict, Optional, Tuple, Callable
from datetime import datetime
from pathlib import Path
import json

# GlobalMOO SDK imports
from globalmoo.client import Client
from globalmoo.credentials import Credentials
from globalmoo.request.create_model import CreateModel
from globalmoo.request.create_project import CreateProject
from globalmoo.request.load_output_cases import LoadOutputCases
from globalmoo.request.load_objectives import LoadObjectives
from globalmoo.request.suggest_inverse import SuggestInverse
from globalmoo.request.load_inversed_output import LoadInversedOutput
from globalmoo.request.initialize_inverse import InitializeInverse
from globalmoo.enums.input_type import InputType
from globalmoo.enums.objective_type import ObjectiveType

logger = logging.getLogger(__name__)


class GlobalMOOOptimizer:
    """
    Wrapper for GlobalMOO API that manages the optimization lifecycle.

    Usage:
        optimizer = GlobalMOOOptimizer(api_key="...")
        optimizer.setup(seed_data)
        optimizer.set_objectives(targets)
        results = optimizer.run_optimization(forward_model, max_iterations=100)
        pareto = optimizer.get_pareto_frontier()
        impacts = optimizer.get_impact_factors()
    """

    def __init__(
        self,
        api_key: Optional[str] = None,
        base_uri: str = "https://api.globalmoo.ai/api",
        results_dir: str = "./storage/results"
    ):
        """
        Initialize the GlobalMOO optimizer.

        Args:
            api_key: GlobalMOO API key. If None, reads from GLOBALMOO_API_KEY env var.
            base_uri: GlobalMOO API endpoint.
            results_dir: Directory to store optimization results.
        """
        self.api_key = api_key or os.getenv("GLOBALMOO_API_KEY")
        if not self.api_key:
            raise ValueError(
                "GlobalMOO API key required. Set GLOBALMOO_API_KEY environment variable "
                "or pass api_key parameter."
            )

        self.credentials = Credentials(api_key=self.api_key, base_uri=base_uri)
        self.client = Client(credentials=self.credentials)

        self.results_dir = Path(results_dir)
        self.results_dir.mkdir(parents=True, exist_ok=True)

        self.vector_codec = VectorCodec()
        self.outcome_codec = OutcomeCodec()

        # State tracking
        self.model_id: Optional[str] = None
        self.project_id: Optional[str] = None
        self.trial_id: Optional[str] = None
        self.objective_id: Optional[str] = None

        # Results storage
        self.all_configs: List[List[int]] = []
        self.all_outcomes: List[List[float]] = []
        self.iteration_log: List[Dict] = []

    def setup(
        self,
        seed_data: List[Tuple[List[int], List[float]]],
        model_name: str = "VERILINGUA_VERIX"
    ) -> None:
        """
        Set up GlobalMOO model, project, and trial with seed data.

        Args:
            seed_data: List of (config_vector, outcome_vector) tuples for initial training.
            GlobalMOO needs seed data to bootstrap its learning.
    model_name: Name for this optimization model.
"""
logger.info(f"Setting up GlobalMOO with {len(seed_data)} seed cases")

# Create model
model_response = self.client.execute_request(CreateModel(
    name=f"{model_name}_{datetime.now().strftime('%Y%m%d_%H%M%S')}"
))
self.model_id = model_response.id
logger.info(f"Created model: {self.model_id}")

# Create project with input bounds
project_response = self.client.execute_request(CreateProject(
    model_id=self.model_id,
    input_count=self.vector_codec.INPUT_COUNT,
    minimums=self.vector_codec.minimums,
    maximums=self.vector_codec.maximums,
    input_types=[InputType.INTEGER] * self.vector_codec.INPUT_COUNT,
    categories=[]  # No categorical variables in our setup
))
self.project_id = project_response.id
logger.info(f"Created project: {self.project_id}")

# Get initial input cases from project (GlobalMOO suggests initial points)
initial_inputs = project_response.input_cases

# If we have fewer seed cases than GlobalMOO suggests, we need to evaluate more
# For now, assume seed_data covers the initial cases
if len(seed_data) < len(initial_inputs):
    logger.warning(
        f"Seed data ({len(seed_data)}) smaller than suggested inputs ({len(initial_inputs)}). "
        "Consider providing more seed data."
    )

# Prepare output cases from seed data
output_cases = [outcome for _, outcome in seed_data]

# Create trial with seed data
trial_response = self.client.execute_request(LoadOutputCases(
    model_id=self.model_id,
    project_id=self.project_id,
    output_count=self.outcome_codec.OUTPUT_COUNT,
    output_cases=output_cases
```

```
        ))
        self.trial_id = trial_response.id
        logger.info(f"Created trial: {self.trial_id}")

        # Store seed data
        for config, outcome in seed_data:
            self.all_configs.append(config)
            self.all_outcomes.append(outcome)

    def set_objectives(
        self,
        objectives: Dict[str, Dict],
        reference_config: Optional[List[int]] = None,
        reference_outcome: Optional[List[float]] = None
    ) -> None:
        """
        Set optimization objectives.

        Args:
            objectives: Dict mapping outcome names to objective specs.
                        Each spec has 'type' and 'value' keys.
                        Example: {"task_accuracy": {"type": "greater_than_equal", "value": 0.85}}
            reference_config: A known-good configuration vector for initialization.
            reference_outcome: The outcomes from the reference configuration.
        """
        # Map objective type strings to GlobalMOO enums
        type_map = {
            "maximize": ObjectiveType.MAXIMIZE,
            "minimize": ObjectiveType.MINIMIZE,
            "greater_than": ObjectiveType.GREATER_THAN,
            "greater_than_equal": ObjectiveType.GREATER_THAN_EQUAL,
            "less_than": ObjectiveType.LESS_THAN,
            "less_than_equal": ObjectiveType.LESS_THAN_EQUAL,
            "equal": ObjectiveType.EQUAL,
            "percent": ObjectiveType.PERCENT,
        }

        # Build objective vectors in the order of our outcome codec
        objective_values = []
        objective_types = []

        for name in self.outcome_codec.OUTPUT_NAMES:
            if name in objectives:
                spec = objectives[name]
            objective_values.append(spec["value"])
            objective_types.append(type_map[spec["type"]])
        else:
            # Default: maximize if higher-is-better, minimize otherwise
            idx = self.outcome_codec.OUTPUT_NAMES.index(name)
            if self.outcome_codec.OUTPUT_DIRECTIONS[idx]:
                objective_types.append(ObjectiveType.MAXIMIZE)
                objective_values.append(1.0)  # Target maximum
            else:
                objective_types.append(ObjectiveType.MINIMIZE)
                objective_values.append(0.0)  # Target minimum

    # Use last seed data point as reference if not provided
    if reference_config is None:
        reference_config = self.all_configs[-1]
    if reference_outcome is None:
        reference_outcome = self.all_outcomes[-1]

    # Define bounds for objectives (how much deviation is acceptable)
    # These are soft bounds for the optimization
    min_bounds = [-0.1] * self.outcome_codec.OUTPUT_COUNT
    max_bounds = [0.1] * self.outcome_codec.OUTPUT_COUNT

    # Load objectives
    objective_response = self.client.execute_request(LoadObjectives(
        model_id=self.model_id,
        project_id=self.project_id,
        trial_id=self.trial_id,
        objectives=objective_values,
        objective_types=objective_types,
        initial_input=reference_config,
        initial_output=reference_outcome,
        minimum_bounds=min_bounds,
        maximum_bounds=max_bounds,
        desired_l1_norm=0.0
    ))
    self.objective_id = objective_response.id
    logger.info(f"Set objectives: {self.objective_id}")

    # Initialize inverse optimization
    self.client.execute_request(InitializeInverse(
        model_id=self.model_id,
        project_id=self.project_id,
        trial_id=self.trial_id,
        objective_id=self.objective_id
    ))
    logger.info("Initialized inverse optimization")

def run_optimization(
    self,
    forward_model: Callable[[List[int]], List[float]],
    max_iterations: int = 100,
    early_stop_threshold: Optional[float] = None
) -> Dict:
    """
    Run the optimization loop.

    Args:
        forward_model: Function that takes a config vector and returns an outcome vector.
        max_iterations: Maximum optimization iterations.
        early_stop_threshold: Stop early if all objectives within this threshold of targets.

    Returns:
        Dict with optimization results including Pareto frontier.
    """
    logger.info(f"Starting optimization loop (max {max_iterations} iterations)")

    for iteration in range(max_iterations):
        # Get next configuration to try from GlobalMOO
        suggest_response = self.client.execute_request(SuggestInverse(
            model_id=self.model_id,
            project_id=self.project_id,
            trial_id=self.trial_id,
            objective_id=self.objective_id
        ))

        suggested_config = suggest_response.input

        # Validate the suggested configuration
        if not self.vector_codec.validate_vector(suggested_config):
            logger.warning(f"Invalid config suggested: {suggested_config}")
            continue

        # Evaluate the configuration using the forward model
        try:
            outcomes = forward_model(suggested_config)
        except Exception as e:
            logger.error(f"Forward model failed: {e}")
            # Return worst-case outcomes on failure
            outcomes = self._worst_case_outcomes()

        # Report results back to GlobalMOO
        inverse_response = self.client.execute_request(LoadInversedOutput(
            model_id=self.model_id,
            project_id=self.project_id,
            trial_id=self.trial_id,
            objective_id=self.objective_id,
            output=outcomes
        ))

        # Store results
        self.all_configs.append(suggested_config)
        self.all_outcomes.append(outcomes)
        self.iteration_log.append({
            "iteration": iteration,
            "config": suggested_config,
            "outcomes": outcomes,
            "timestamp": datetime.now().isoformat()
        })

        # Log progress
        if iteration % 10 == 0:
            logger.info(f"Iteration {iteration}: outcomes = {outcomes[:4]}...")

        # Check for early stopping
        if inverse_response.should_stop():
            logger.info(f"GlobalMOO signaled stop at iteration {iteration}")
            break

        if early_stop_threshold and self._check_early_stop(outcomes, early_stop_threshold):
            logger.info(f"Early stop threshold met at iteration {iteration}")
            break

    # Compute final results
    results = self._compile_results()

    # Save results to disk
    self._save_results(results)

    return results

def get_pareto_frontier(self) -> List[Tuple[List[int], List[float]]]:
    """
    Compute the Pareto frontier from all evaluated configurations.

    Returns:
        List of (config_vector, outcome_vector) tuples that are Pareto optimal.
    """
    n = len(self.all_outcomes)
    is_pareto = [True] * n

    for i in range(n):
        if not is_pareto[i]:
            continue

        for j in range(n):
            if i == j or not is_pareto[j]:
                continue

            # Check if j dominates i
            if self._dominates(self.all_outcomes[j], self.all_outcomes[i]):
                is_pareto[i] = False
                break

    return [
        (self.all_configs[i], self.all_outcomes[i])
        for i in range(n) if is_pareto[i]
    ]

def _dominates(self, outcome_a: List[float], outcome_b: List[float]) -> bool:
    """
    Check if outcome_a dominates outcome_b.

    A dominates B if A is at least as good on all objectives
    and strictly better on at least one.
    """
    dominated_on_all = True
    strictly_better_on_one = False

    for i, (a, b) in enumerate(zip(outcome_a, outcome_b)):
        if self.outcome_codec.OUTPUT_DIRECTIONS[i]:
            # Higher is better
            if a < b:
                dominated_on_all = False
            if a > b:
                strictly_better_on_one = True
        else:
            # Lower is better
            if a > b:
                dominated_on_all = False
            if a < b:
                strictly_better_on_one = True

    return dominated_on_all and strictly_better_on_one

def get_impact_factors(self) -> Dict[str, Dict[str, float]]:
    """
    Compute impact factors: how much each input affects each output.

    Uses Pearson correlation as a simple impact measure.
    For more sophisticated analysis, GlobalMOO provides native impact computation.

    Returns:
        Dict mapping input names to dict of output impacts.
    """
    import numpy as np

    config_matrix = np.array(self.all_configs)
    outcome_matrix = np.array(self.all_outcomes)

    input_names = [
        "russian_aspect", "turkish_evidential", "arabic_morphology",
        "german_composition", "japanese_keigo", "chinese_classifiers",
        "guugu_spatial", "max_frames", "verix_strictness",
        "confidence_required", "ground_required", "activation_style",
        "verbosity", "fewshot_count", "show_legend", "prompt_length"
    ]

    impacts = {}

    for i, input_name in enumerate(input_names):
        impacts[input_name] = {}
        input_values = config_matrix[:, i]

        for j, output_name in enumerate(self.outcome_codec.OUTPUT_NAMES):
            output_values = outcome_matrix[:, j]

            # Compute correlation
            if np.std(input_values) > 0 and np.std(output_values) > 0:
                corr = np.corrcoef(input_values, output_values)[0, 1]
                impacts[input_name][output_name] = float(corr) if np.isfinite(corr) else 0.0
            else:
                impacts[input_name][output_name] = 0.0

    return impacts

def inverse_query(
    self,
    targets: Dict[str, Dict]
) -> Optional[List[int]]:
    """
    Find a configuration that meets the specified targets.

    Args:
        targets: Dict mapping outcome names to target specs.
                Example: {"task_accuracy": {"type": "greater_than_equal", "value": 0.9}}

    Returns:
        Configuration vector that meets targets, or None if not achievable.
    """
    # This uses the already-configured objectives
    # In practice, you might want to reconfigure objectives for specific queries

    # For now, scan the Pareto frontier for configurations meeting the targets
    pareto = self.get_pareto_frontier()

    for config, outcomes in pareto:
        if self._meets_targets(outcomes, targets):
            return config

    return None

def _meets_targets(self, outcomes: List[float], targets: Dict[str, Dict]) -> bool:
    """Check if outcomes meet all targets."""
    for name, spec in targets.items():
        idx = self.outcome_codec.OUTPUT_NAMES.index(name)
        value = outcomes[idx]
        target_value = spec["value"]
        target_type = spec["type"]

        if target_type == "greater_than" and not (value > target_value):
            return False
        elif target_type == "greater_than_equal" and not (value >= target_value):
            return False
        elif target_type == "less_than" and not (value < target_value):
            return False
        elif target_type == "less_than_equal" and not (value <= target_value):
            return False
        elif target_type == "equal" and not (abs(value - target_value) < 0.001):
            return False

    return True

def _worst_case_outcomes(self) -> List[float]:
    """Return worst-case outcomes for failed evaluations."""
    worst = []
    for i, direction in enumerate(self.outcome_codec.OUTPUT_DIRECTIONS):
        if direction:
            worst.append(0.0)  # Worst for higher-is-better
        else:
            if i == 7:  # tokens_total
                worst.append(10000.0)
            elif i == 8:  # latency_seconds
                worst.append(30.0)
            else:
                worst.append(1.0)  # Worst for lower-is-better (0-1 scale)
    return worst

def _check_early_stop(self, outcomes: List[float], threshold: float) -> bool:
    """Check if all objectives are within threshold of targets."""
    # Implementation depends on how you want to define "close enough"
    return False

def _compile_results(self) -> Dict:
    """Compile optimization results into a summary."""
    pareto = self.get_pareto_frontier()
    impacts = self.get_impact_factors()

    # Find extreme points on Pareto frontier
    extremes = {}
    for i, name in enumerate(self.outcome_codec.OUTPUT_NAMES):
        direction = self.outcome_codec.OUTPUT_DIRECTIONS[i]
        if direction:
            best_idx = max(range(len(pareto)), key=lambda x: pareto[x][1][i])
        else:
            best_idx = min(range(len(pareto)), key=lambda x: pareto[x][1][i])

        config, outcomes = pareto[best_idx]
```

```
            extremes[name] = {
                "config": config,
                "value": outcomes[i],
                "all_outcomes": dict(zip(self.outcome_codec.OUTPUT_NAMES, outcomes))
            }

        # Identify high-impact variables
        high_impact = []
        for input_name, output_impacts in impacts.items():
            for output_name, impact in output_impacts.items():
                if abs(impact) > 0.3:
                    high_impact.append({
                        "input": input_name,
                        "output": output_name,
                        "impact": impact
                    })
        high_impact.sort(key=lambda x: -abs(x["impact"]))

        return {
            "iterations": len(self.all_outcomes) - len(self.all_configs),  # Subtract seed
            "pareto_size": len(pareto),
            "pareto_configs": [p[0] for p in pareto],
            "pareto_outcomes": [p[1] for p in pareto],
            "extremes": extremes,
            "high_impact_variables": high_impact[:10],
            "impacts": impacts
        }

    def _save_results(self, results: Dict) -> None:
        """Save results to disk."""
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

        # Save full results
        results_path = self.results_dir / f"optimization_{timestamp}.json"
        with open(results_path, 'w') as f:
            json.dump(results, f, indent=2)

        # Save iteration log
        log_path = self.results_dir / f"iteration_log_{timestamp}.jsonl"
        with open(log_path, 'w') as f:
            for entry in self.iteration_log:
                f.write(json.dumps(entry) + "\n")

        logger.info(f"Results saved to {results_path}")
    def close(self) -> None:
        """Clean up client resources."""
        if hasattr(self.client, 'http_client'):
            self.client.http_client.close()
```

3.5 The evaluation engine

The forward model needs to actually run your cognitive architecture and measure outcomes.

```
python
```

```
import time
from typing import List
import dspy

class EvaluationEngine:
    """
    Implements the forward model for GlobalMOO optimization.

    This is where configuration vectors become measured outcomes.
    """

    def __init__(
        self,
        task_corpus: List,
        lm: dspy.LM,
        prompt_builder,
        verix_parser,
        cluster_manager=None  # DSPy cluster manager for optimized prompts
    ):
        self.task_corpus = task_corpus
        self.lm = lm
        self.prompt_builder = prompt_builder
        self.verix_parser = verix_parser
        self.cluster_manager = cluster_manager
        self.vector_codec = VectorCodec()
        self.outcome_codec = OutcomeCodec()

    def evaluate(self, config_vector: List[int]) -> List[float]:
        """
        The forward model: config vector -> outcome vector.

        This is what GlobalMOO calls to evaluate a configuration.
        """
        # Decode configuration
        config = self.vector_codec.from_vector(config_vector)

        # Get prompts (use DSPy-optimized if available)
        if self.cluster_manager:
            module = self.cluster_manager.get_module_for_config(config)
        else:
            module = None

        # Evaluate on task corpus
        results = []
        for task in self.task_corpus:
            task_result = self._evaluate_single_task(config, task, module)
            results.append(task_result)

        # Aggregate outcomes
        aggregated = self._aggregate_outcomes(results)

        # Encode as vector
        return self.outcome_codec.to_vector(aggregated)

    def _evaluate_single_task(self, config, task, module=None) -> Outcomes:
        """Evaluate a single task and measure outcomes."""
        start_time = time.time()

        try:
            # Generate response
            if module:
                with dspy.context(lm=self.lm):
                    prediction = module(task=task.prompt)
                response = prediction.analysis
            else:
                system, user = self.prompt_builder.build(task.prompt, task.category, config)
                response = self._call_llm(system, user)

            latency = time.time() - start_time

            # Measure outcomes
            return Outcomes(
                task_accuracy=self._measure_accuracy(response, task),
                math_accuracy=self._measure_math_accuracy(response, task),
                calibration_score=self._measure_calibration(response, task),
                format_compliance=self._measure_format(response, config),
                ground_chain_validity=self._measure_grounding(response),
                focus_score=self._measure_focus(response, task),
                nuance_appropriateness=self._measure_nuance(response, task),
                tokens_total=self._count_tokens(response),
                latency_seconds=latency,
                edge_case_robustness=self._measure_robustness(response, task),
                epistemic_consistency=self._measure_consistency(response)
            )

        except Exception as e:
            logger.error(f"Task evaluation failed: {e}")
        return self._worst_case_task_outcome()

    def _call_llm(self, system: str, user: str) -> str:
        """Call the LLM with system and user prompts."""
        response = self.lm(
            messages=[{"role": "user", "content": user}],
            system=system
        )
        return response[0] if response else ""

    def _measure_accuracy(self, response: str, task) -> float:
        """Measure task accuracy against ground truth if available."""
        if hasattr(task, 'reference') and task.reference:
            # Compare response to reference
            # Implementation depends on task type
            return self._semantic_similarity(response, task.reference)
        else:
            # For open-ended tasks, use a heuristic or return neutral score
            return 0.5 if len(response) > 50 else 0.0

    def _measure_math_accuracy(self, response: str, task) -> float:
        """Measure mathematical accuracy for math tasks."""
        if task.category != "math":
            return 1.0  # Not applicable

        if hasattr(task, 'reference') and task.reference:
            # Extract numerical answer and compare
            # This is simplified—real implementation would parse numbers
            return 1.0 if task.reference in response else 0.0
        return 0.5

    def _measure_calibration(self, response: str, task) -> float:
        """
        Measure calibration using Brier score.

        Returns lower scores for better calibration (0 = perfect).
        """
        claims = self.verix_parser.parse(response)
        if not claims:
            return 0.5  # No claims to calibrate

        # For each claim with stated confidence, compare to actual correctness
        brier_scores = []
        for claim in claims:
            if claim.confidence is not None:
                # Determine if claim is correct (simplified)
                correct = self._is_claim_correct(claim, task)
                brier_scores.append((claim.confidence - float(correct)) ** 2)

        return sum(brier_scores) / len(brier_scores) if brier_scores else 0.5

    def _measure_format(self, response: str, config) -> float:
        """Measure VERIX format compliance."""
        return self.verix_parser.format_compliance(response)

    def _measure_grounding(self, response: str) -> float:
        """Measure ground chain validity."""
        claims = self.verix_parser.parse(response)
        if not claims:
            return 0.5

        grounded_claims = [c for c in claims if c.ground]
        if not grounded_claims:
            return 0.0

        # Check if grounds are plausible (simplified)
        valid_grounds = sum(1 for c in grounded_claims if len(c.ground) > 3)
        return valid_grounds / len(grounded_claims)

    def _measure_focus(self, response: str, task) -> float:
        """Measure how focused the response is on the task."""
        # Simplified: check response length relative to expected
        expected_length = getattr(task, 'expected_length', 500)
        actual_length = len(response)

        if actual_length < expected_length * 0.3:
            return 0.3  # Too short
        elif actual_length > expected_length * 3:
            return 0.5  # Too long
        else:
            return 1.0

    def _measure_nuance(self, response: str, task) -> float:
        """Measure nuance appropriateness."""
        complexity = getattr(task, 'complexity', 2)

        # Count nuance markers
        nuance_phrases = ["however", "although", "on the other hand", "it depends", "nuanced"]
        nuance_count = sum(1 for p in nuance_phrases if p in response.lower())

        # Match nuance to complexity
        if complexity >= 3 and nuance_count >= 2:
            return 1.0
        elif complexity <= 1 and nuance_count <= 1:
            return 1.0
        elif abs(complexity - nuance_count) <= 1:
            return 0.7
        else:
            return 0.4

    def _count_tokens(self, response: str) -> float:
        """Count tokens in response."""
        # Simplified: estimate tokens as words * 1.3
        return len(response.split()) * 1.3

    def _measure_robustness(self, response: str, task) -> float:
        """Measure edge case robustness."""
        if getattr(task, 'is_edge_case', False):
            # Check for appropriate hedging
            hedging_phrases = ["uncertain", "unclear", "depends", "might", "could"]
            has_hedging = any(p in response.lower() for p in hedging_phrases)
            return 0.8 if has_hedging else 0.3
        return 0.7  # Default for non-edge cases

    def _measure_consistency(self, response: str) -> float:
        """Measure epistemic consistency."""
        claims = self.verix_parser.parse(response)
        if len(claims) < 2:
            return 1.0  # Can't be inconsistent with one claim

        # Check for contradictory high-confidence claims
        high_conf = [c for c in claims if c.confidence and c.confidence > 0.8]

        # Simplified: check if any high-confidence claims seem contradictory
        # Real implementation would need semantic analysis
        return 1.0 if len(high_conf) <= 3 else 0.8

    def _aggregate_outcomes(self, results: List[Outcomes]) -> Outcomes:
        """Aggregate outcomes across multiple tasks."""
        if not results:
            return self._worst_case_task_outcome()
        return Outcomes(
            task_accuracy=sum(r.task_accuracy for r in results) / len(results),
            math_accuracy=sum(r.math_accuracy for r in results) / len(results),
            calibration_score=sum(r.calibration_score for r in results) / len(results),
            format_compliance=sum(r.format_compliance for r in results) / len(results),
            ground_chain_validity=sum(r.ground_chain_validity for r in results) / len(results),
            focus_score=sum(r.focus_score for r in results) / len(results),
            nuance_appropriateness=sum(r.nuance_appropriateness for r in results) / len(results),
            tokens_total=sum(r.tokens_total for r in results),  # Sum, not average
            latency_seconds=sum(r.latency_seconds for r in results),  # Sum, not average
            edge_case_robustness=sum(r.edge_case_robustness for r in results) / len(results),
            epistemic_consistency=sum(r.epistemic_consistency for r in results) / len(results),
        )

    def _worst_case_task_outcome(self) -> Outcomes:
        """Return worst-case outcomes for failed tasks."""
        return Outcomes(
            task_accuracy=0.0,
            math_accuracy=0.0,
            calibration_score=1.0,
            format_compliance=0.0,
            ground_chain_validity=0.0,
            focus_score=0.0,
            nuance_appropriateness=0.0,
            tokens_total=10000,
            latency_seconds=30.0,
            edge_case_robustness=0.0,
            epistemic_consistency=0.0
        )

    def _semantic_similarity(self, response: str, reference: str) -> float:
        """Compute semantic similarity between response and reference."""
        # Simplified: use word overlap
        response_words = set(response.lower().split())
        reference_words = set(reference.lower().split())

        if not reference_words:
            return 0.0

        overlap = len(response_words & reference_words)
        return overlap / len(reference_words)

    def _is_claim_correct(self, claim, task) -> bool:
        """Determine if a claim is correct."""
```

```python
    # This is a placeholder—real implementation needs task-specific logic
    return True
```

---

4) The Three-MOO Cascade

The Developer Integration Guide describes a three-phase optimization approach. Here's how to implement each phase with GlobalMOO.

4.1 Phase A: Framework structure optimization

Phase A decides what the framework should contain—not the parameter values, but the structure itself.

python

```
class PhaseAOptimizer:
    """
    Phase A: Optimize framework structure.

    This phase answers: What should VERILINGUA × VERIX contain?
    - Which frames should be in the vocabulary?
    - What routing rule complexity is appropriate?
    - What VERIX strictness levels should be options?
    """

    def __init__(self, evaluation_engine: EvaluationEngine):
        self.engine = evaluation_engine

    def run(
        self,
        candidate_structures: List[Dict],
        evaluation_tasks: List
    ) -> Dict:
        """
        Evaluate candidate framework structures.

        Args:
            candidate_structures: List of structure definitions
            evaluation_tasks: Tasks to evaluate on

        Returns:
            Analysis of each structure including recommended choice
        """
        results = []

        for structure in candidate_structures:
            # Convert structure to configurations
            configs = self._structure_to_configs(structure)

            # Evaluate each config
            outcomes = []
            for config in configs:
                vector = VectorCodec().to_vector(config)
                outcome = self.engine.evaluate(vector)
                outcomes.append(outcome)

            # Compute structure-level metrics
            structure_result = {
```

```
            "structure": structure,
            "expressiveness": self._measure_expressiveness(outcomes),
            "parsimony": self._measure_parsimony(structure, outcomes),
            "stability": self._measure_stability(outcomes)
        }
        results.append(structure_result)

    # Rank structures
    results.sort(key=lambda x: (
        x["expressiveness"] + x["parsimony"] + x["stability"]
    ), reverse=True)

    return {
        "structures_evaluated": len(results),
        "rankings": results,
        "recommended": results[0]["structure"]
    }

def _structure_to_configs(self, structure: Dict) -> List[FullConfig]:
    """Generate concrete configurations from a structure definition."""
    configs = []

    # Generate variations to test stability
    for max_frames in [1, 2, 3]:
        for verix_strictness in [VERIXStrictness.PARTIAL, VERIXStrictness.FULL]:
            config = FullConfig(
                framework=FrameworkConfig(
                    russian_aspect="russian_aspect" in structure.get("frames", []),
                    turkish_evidential="turkish_evidential" in structure.get("frames", []),
                    arabic_morphology="arabic_morphology" in structure.get("frames", []),
                    german_composition="german_composition" in structure.get("frames", []),
                    japanese_keigo="japanese_keigo" in structure.get("frames", []),
                    chinese_classifiers="chinese_classifiers" in structure.get("frames", []),
                    guugu_spatial="guugu_spatial" in structure.get("frames", []),
                    max_frames_per_task=max_frames,
                    verix_strictness=verix_strictness,
                    verix_confidence_required=True,
                    verix_ground_chain_required=structure.get("require_grounding", False)
                ),
                prompt=PromptConfig(
                    activation_style=ActivationStyle.EXPLICIT,
                    activation_verbosity=1,
                    fewshot_count=1,
                    show_verix_legend=True,
                system_prompt_length=1
            )
        )
        configs.append(config)

    return configs

def _measure_expressiveness(self, outcomes: List[List[float]]) -> float:
    """Measure best achievable performance with this structure."""
    if not outcomes:
        return 0.0

    # Best task accuracy across all configs
    best_accuracy = max(o[0] for o in outcomes)  # Index 0 = task_accuracy

    # Best calibration (lower is better, so min)
    best_calibration = min(o[2] for o in outcomes)  # Index 2 = calibration

    return best_accuracy * (1 - best_calibration)

def _measure_parsimony(self, structure: Dict, outcomes: List[List[float]]) -> float:
    """Measure efficiency: performance per unit complexity."""
    complexity = len(structure.get("frames", []))
    if complexity == 0:
        return 0.0

    avg_accuracy = sum(o[0] for o in outcomes) / len(outcomes)
    return avg_accuracy / (1 + complexity * 0.1)

def _measure_stability(self, outcomes: List[List[float]]) -> float:
    """Measure consistency across configuration variations."""
    if len(outcomes) < 2:
        return 1.0

    import numpy as np
    accuracies = [o[0] for o in outcomes]
    variance = np.var(accuracies)

    return 1.0 / (1 + variance * 10)
```

4.2 Phase B: Edge discovery

Phase B finds the boundaries of the feasible cognitive space—where performance degrades sharply.

python

```python
class PhaseBEdgeDiscovery:
    """
    Phase B: Discover edges of the cognitive space.

    This phase answers: Where are the boundaries?
    - Where does performance collapse?
    - What configuration changes cause sharp degradation?
    - What is the stability radius around good configurations?
    """

    def __init__(self, optimizer: GlobalMOOOptimizer):
        self.optimizer = optimizer

    def run(
        self,
        seed_configs: List[List[int]],
        probe_iterations: int = 50
    ) -> Dict:
        """
        Discover edges through targeted probing.

        Args:
            seed_configs: Known-good configurations to start from
            probe_iterations: How many edge probes to run

        Returns:
            Edge analysis including stability radii and failure modes
        """
        edges = []

        for seed in seed_configs:
            # Probe around this seed configuration
            seed_edges = self._probe_edges(seed, probe_iterations // len(seed_configs))
            edges.extend(seed_edges)

        # Analyze discovered edges
        stability_radii = self._compute_stability_radii(seed_configs, edges)
        failure_modes = self._categorize_failure_modes(edges)

        # Compute recommended bounds
        bounds = self._compute_safe_bounds(seed_configs, stability_radii)

        return {
            "edges_discovered": len(edges),
            "stability_radii": stability_radii,
            "failure_modes": failure_modes,
            "recommended_bounds": bounds
        }

    def _probe_edges(self, seed: List[int], n_probes: int) -> List[Dict]:
        """Probe for edges around a seed configuration."""
        edges = []
        codec = VectorCodec()

        for _ in range(n_probes):
            # Perturb seed configuration
            perturbed = seed.copy()

            # Change 1-3 variables
            import random
            n_changes = random.randint(1, 3)
            positions = random.sample(range(len(seed)), n_changes)

            for pos in positions:
                min_val, max_val = codec.BOUNDS[pos]
                perturbed[pos] = random.randint(min_val, max_val)

            # Evaluate perturbed configuration
            outcome = self.optimizer.all_outcomes[-1] if self.optimizer.all_outcomes else None
            if outcome:
                # Check if this is an edge (significant performance drop)
                seed_outcome = self._get_seed_outcome(seed)
                if seed_outcome and self._is_edge(seed_outcome, outcome):
                    edges.append({
                        "seed": seed,
                        "perturbed": perturbed,
                        "changed_positions": positions,
                        "performance_delta": self._compute_delta(seed_outcome, outcome)
                    })

        return edges

    def _get_seed_outcome(self, seed: List[int]) -> Optional[List[float]]:
        """Get the outcome for a seed configuration."""
        for i, config in enumerate(self.optimizer.all_configs):
            if config == seed:
                return self.optimizer.all_outcomes[i]
```

```python
        return None

    def _is_edge(self, seed_outcome: List[float], perturbed_outcome: List[float]) -> bool:
        """Check if the perturbation caused an edge (significant degradation)."""
        # Define edge as >20% degradation in any key metric
        key_indices = [0, 2, 9, 10]  # accuracy, calibration, robustness, consistency

        for idx in key_indices:
            seed_val = seed_outcome[idx]
            perturbed_val = perturbed_outcome[idx]

            direction = OutcomeCodec.OUTPUT_DIRECTIONS[idx]
            if direction:  # Higher is better
                if seed_val > 0 and (seed_val - perturbed_val) / seed_val > 0.2:
                    return True
            else:  # Lower is better
                if seed_val > 0 and (perturbed_val - seed_val) / seed_val > 0.2:
                    return True

        return False

    def _compute_delta(self, seed_outcome: List[float], perturbed_outcome: List[float]) -> Dict:
        """Compute the performance delta between outcomes."""
        return {
            name: perturbed_outcome[i] - seed_outcome[i]
            for i, name in enumerate(OutcomeCodec.OUTPUT_NAMES)
        }

    def _compute_stability_radii(
        self,
        seeds: List[List[int]],
        edges: List[Dict]
    ) -> Dict[str, float]:
        """Compute stability radius per configuration variable."""
        radii = {}
        codec = VectorCodec()

        for var_idx in range(codec.INPUT_COUNT):
            var_name = [
                "russian_aspect", "turkish_evidential", "arabic_morphology",
                "german_composition", "japanese_keigo", "chinese_classifiers",
                "guugu_spatial", "max_frames", "verix_strictness",
                "confidence_required", "ground_required", "activation_style",
                "verbosity", "fewshot_count", "show_legend", "prompt_length"
            ][var_idx]

            # Find edges caused by changing this variable
            var_edges = [e for e in edges if var_idx in e["changed_positions"]]

            if var_edges:
                # Radius = average change size that caused an edge
                changes = []
                for edge in var_edges:
                    seed_val = edge["seed"][var_idx]
                    perturbed_val = edge["perturbed"][var_idx]
                    changes.append(abs(perturbed_val - seed_val))

                radii[var_name] = sum(changes) / len(changes)
            else:
                # No edges found for this variable—consider it stable
                min_val, max_val = codec.BOUNDS[var_idx]
                radii[var_name] = max_val - min_val  # Full range

        return radii

    def _categorize_failure_modes(self, edges: List[Dict]) -> Dict[str, int]:
        """Categorize the types of failures encountered."""
        categories = {
            "accuracy_collapse": 0,
            "calibration_failure": 0,
            "robustness_failure": 0,
            "consistency_failure": 0,
            "efficiency_failure": 0
        }

        for edge in edges:
            delta = edge["performance_delta"]

            if delta.get("task_accuracy", 0) < -0.2:
                categories["accuracy_collapse"] += 1
            if delta.get("calibration_score", 0) > 0.2:
                categories["calibration_failure"] += 1
            if delta.get("edge_case_robustness", 0) < -0.2:
                categories["robustness_failure"] += 1
            if delta.get("epistemic_consistency", 0) < -0.2:
                categories["consistency_failure"] += 1
            if delta.get("tokens_total", 0) > 500:
                categories["efficiency_failure"] += 1
```

```python
    return categories

def _compute_safe_bounds(
    self,
    seeds: List[List[int]],
    stability_radii: Dict[str, float]
) -> Dict:
    """Compute safe bounds for Phase C based on discovered edges."""
    codec = VectorCodec()

    # Start with full bounds
    safe_mins = codec.minimums.copy()
    safe_maxs = codec.maximums.copy()

    # Tighten bounds based on stability radii
    # Variables with small stability radii should be constrained
    var_names = [
        "russian_aspect", "turkish_evidential", "arabic_morphology",
        "german_composition", "japanese_keigo", "chinese_classifiers",
        "guugu_spatial", "max_frames", "verix_strictness",
        "confidence_required", "ground_required", "activation_style",
        "verbosity", "fewshot_count", "show_legend", "prompt_length"
    ]

    for i, var_name in enumerate(var_names):
        radius = stability_radii.get(var_name, 10)

        if radius < 1:
            # Very unstable variable—constrain to seed values
            seed_values = [s[i] for s in seeds]
            safe_mins[i] = min(seed_values)
            safe_maxs[i] = max(seed_values)

    return {
        "minimums": safe_mins,
        "maximums": safe_maxs,
        "constrained_variables": [
            var_names[i] for i in range(len(var_names))
            if safe_mins[i] != codec.minimums[i] or safe_maxs[i] != codec.maximums[i]
        ]
    }
```

4.3 Phase C: Production optimization

Phase C runs full multi-objective optimization within the bounds discovered by Phase B.

python

```python
def run_phase_c(
    evaluation_engine: EvaluationEngine,
    phase_b_bounds: Dict,
    seed_data: List[Tuple[List[int], List[float]]],
    objectives: Dict[str, Dict],
    max_iterations: int = 100
) -> Dict:
    """
    Run Phase C production optimization.

    Args:
        evaluation_engine: The forward model
        phase_b_bounds: Safe bounds from Phase B
        seed_data: Initial configurations and outcomes
        objectives: Optimization objectives
        max_iterations: Maximum iterations

    Returns:
        Optimization results including Pareto frontier
    """
    # Create optimizer with constrained bounds
    optimizer = GlobalMOOOptimizer()

    # Override bounds with Phase B constraints
    optimizer.vector_codec.BOUNDS = [
        (phase_b_bounds["minimums"][i], phase_b_bounds["maximums"][i])
        for i in range(len(phase_b_bounds["minimums"]))
    ]

    # Setup and run
    optimizer.setup(seed_data)
    optimizer.set_objectives(objectives)

    results = optimizer.run_optimization(
        forward_model=evaluation_engine.evaluate,
        max_iterations=max_iterations
    )

    return results
```

5) Mode distillation

After optimization, distill the Pareto frontier into named modes.

```
python
```

```python
from dataclasses import dataclass
from typing import Dict, List, Optional

@dataclass
class CognitiveMode:
    """A named configuration preset."""
    name: str
    description: str
    config: FullConfig
    optimized_for: List[str]  # Which objectives this mode prioritizes
    trade_offs: Dict[str, str]  # What it sacrifices


class ModeDistiller:
    """
    Distill Pareto frontier into named modes.
    """

    def __init__(self, pareto_configs: List[List[int]], pareto_outcomes: List[List[float]]):
        self.configs = pareto_configs
        self.outcomes = pareto_outcomes
        self.codec = VectorCodec()
        self.outcome_codec = OutcomeCodec()

    def distill(self) -> List[CognitiveMode]:
        """
        Create named modes from Pareto frontier.

        Returns:
            List of distilled modes
        """
        modes = []

        # Audit Mode: best calibration + consistency
        audit_idx = self._find_best_for(["calibration_score", "epistemic_consistency"],
                                        directions=[False, True])
        modes.append(CognitiveMode(
            name="audit",
            description="Maximum transparency and calibration for high-stakes decisions",
            config=self.codec.from_vector(self.configs[audit_idx]),
            optimized_for=["calibration", "consistency", "grounding"],
            trade_offs={"tokens": "higher", "latency": "higher"}
        ))
```

```
# Speed Mode: best tokens + latency
speed_idx = self._find_best_for(["tokens_total", "latency_seconds"],
                                directions=[False, False])
modes.append(CognitiveMode(
    name="speed",
    description="Minimum overhead for quick responses",
    config=self.codec.from_vector(self.configs[speed_idx]),
    optimized_for=["tokens", "latency"],
    trade_offs={"format_compliance": "lower", "grounding": "lower"}
))

# Research Mode: best accuracy + robustness
research_idx = self._find_best_for(["task_accuracy", "edge_case_robustness"],
                                   directions=[True, True])
modes.append(CognitiveMode(
    name="research",
    description="Maximum accuracy and robustness for complex analysis",
    config=self.codec.from_vector(self.configs[research_idx]),
    optimized_for=["accuracy", "robustness", "nuance"],
    trade_offs={"tokens": "higher", "latency": "higher"}
))

# Math Mode: best math_accuracy + calibration
math_idx = self._find_best_for(["math_accuracy", "calibration_score"],
                               directions=[True, False])
modes.append(CognitiveMode(
    name="math",
    description="Rigorous calculation with verification steps",
    config=self.codec.from_vector(self.configs[math_idx]),
    optimized_for=["math_accuracy", "calibration"],
    trade_offs={"latency": "higher"}
))

# Synthesis Mode: balanced across all objectives
synthesis_idx = self._find_most_balanced()
modes.append(CognitiveMode(
    name="synthesis",
    description="Balanced performance for document creation",
    config=self.codec.from_vector(self.configs[synthesis_idx]),
    optimized_for=["balance", "focus", "nuance"],
    trade_offs={"extremes": "avoided"}
))
```

```
        return modes

    def _find_best_for(
        self,
        outcome_names: List[str],
        directions: List[bool]
    ) -> int:
        """Find config that best optimizes given outcomes."""
        scores = []

        for i, outcome in enumerate(self.outcomes):
            score = 0
            for name, direction in zip(outcome_names, directions):
                idx = self.outcome_codec.OUTPUT_NAMES.index(name)
                value = outcome[idx]

                # Normalize to 0-1 range (approximate)
                if direction:
                    score += value
                else:
                    score += (1 - value) if value <= 1 else (1 / value)

            scores.append(score)

        return scores.index(max(scores))

    def _find_most_balanced(self) -> int:
        """Find config with most balanced performance across all objectives."""
        scores = []

        for outcome in self.outcomes:
            # Compute balance as inverse of variance across normalized scores
            normalized = []
            for i, val in enumerate(outcome):
                direction = self.outcome_codec.OUTPUT_DIRECTIONS[i]
                if direction:
                    normalized.append(val)
                else:
                    normalized.append(1 - val if val <= 1 else 1 / val)

            import numpy as np
            variance = np.var(normalized)
            scores.append(1 / (1 + variance))
        return scores.index(max(scores))

    def export_modes(self, path: str) -> None:
        """Export modes to YAML for deployment."""
        import yaml

        modes = self.distill()

        export = {
            "modes": {
                mode.name: {
                    "description": mode.description,
                    "config_vector": self.codec.to_vector(mode.config),
                    "optimized_for": mode.optimized_for,
                    "trade_offs": mode.trade_offs
                }
                for mode in modes
            }
        }

        with open(path, 'w') as f:
            yaml.dump(export, f, default_flow_style=False)
```

---

6) Practical considerations

6.1 Managing API costs

GlobalMOO charges per evaluation. Manage costs by:

```
python
```

```
# Batch tasks per evaluation to amortize API overhead
TASKS_PER_EVALUATION = 10  # Instead of full corpus every time

# Use smaller task corpus during exploration, full corpus for final validation
exploration_corpus = task_corpus[:10]
validation_corpus = task_corpus

# Cache evaluations to avoid re-running identical configs
from functools import lru_cache

@lru_cache(maxsize=1000)
def cached_evaluate(config_tuple):
    config_vector = list(config_tuple)
    return evaluation_engine.evaluate(config_vector)

def evaluate_with_cache(config_vector):
    return cached_evaluate(tuple(config_vector))
```

6.2 Handling failed evaluations

LLM calls can fail. Handle gracefully:

```python
def robust_forward_model(config_vector: List[int], max_retries: int = 3) -> List[float]:
    """Forward model with retry logic."""
    for attempt in range(max_retries):
        try:
            return evaluation_engine.evaluate(config_vector)
        except Exception as e:
            logger.warning(f"Attempt {attempt + 1} failed: {e}")
            if attempt == max_retries - 1:
                logger.error("All retries failed, returning worst-case outcomes")
                return OutcomeCodec().to_vector(evaluation_engine._worst_case_task_outcome())
            time.sleep(2 ** attempt)  # Exponential backoff
```

6.3 Monitoring optimization progress

Track progress to detect issues early:

```python
```

```
class OptimizationMonitor:
    """Monitor optimization progress and detect issues."""

    def __init__(self):
        self.pareto_sizes = []
        self.best_accuracy = []
        self.best_calibration = []

    def record(self, optimizer: GlobalMOOOptimizer) -> None:
        """Record current state."""
        pareto = optimizer.get_pareto_frontier()

        self.pareto_sizes.append(len(pareto))

        if pareto:
            accuracies = [p[1][0] for p in pareto]
            calibrations = [p[1][2] for p in pareto]
            self.best_accuracy.append(max(accuracies))
            self.best_calibration.append(min(calibrations))

    def check_progress(self) -> Dict[str, bool]:
        """Check if optimization is making progress."""
        issues = {}

        # Pareto frontier should grow
        if len(self.pareto_sizes) > 10:
            recent = self.pareto_sizes[-10:]
            if max(recent) == min(recent):
                issues["pareto_stalled"] = True

        # Best metrics should improve
        if len(self.best_accuracy) > 20:
            early = sum(self.best_accuracy[:10]) / 10
            recent = sum(self.best_accuracy[-10:]) / 10
            if recent <= early:
                issues["accuracy_not_improving"] = True

        return issues
```

6.4 Holdout validation

Always validate on data the optimizer hasn't seen:

python

```python
def validate_pareto(
    pareto_configs: List[List[int]],
    holdout_tasks: List,
    evaluation_engine: EvaluationEngine
) -> Dict:
    """Validate Pareto configurations on holdout tasks."""
    original_corpus = evaluation_engine.task_corpus
    evaluation_engine.task_corpus = holdout_tasks

    results = []
    for config in pareto_configs:
        outcome = evaluation_engine.evaluate(config)
        results.append({
            "config": config,
            "holdout_outcomes": outcome
        })

    evaluation_engine.task_corpus = original_corpus

    # Check for overfitting
    training_outcomes = [...]  # From optimization
    holdout_outcomes = [r["holdout_outcomes"] for r in results]

    overfitting_detected = False
    for i, name in enumerate(OutcomeCodec.OUTPUT_NAMES):
        training_avg = sum(o[i] for o in training_outcomes) / len(training_outcomes)
        holdout_avg = sum(o[i] for o in holdout_outcomes) / len(holdout_outcomes)

        direction = OutcomeCodec.OUTPUT_DIRECTIONS[i]
        if direction:
            if holdout_avg < training_avg * 0.8:
                overfitting_detected = True
        else:
            if holdout_avg > training_avg * 1.2:
                overfitting_detected = True

    return {
        "results": results,
        "overfitting_detected": overfitting_detected
    }
```

7) Integration checklist

Before deploying GlobalMOO integration:

1. **API key configured** — Set GLOBALMOO_API_KEY environment variable
2. **Vector codecs implemented** — Stable, versioned encoding for configs and outcomes
3. **Forward model tested** — Evaluation engine returns consistent results
4. **Seed data prepared** — At least 10-20 initial evaluations
5. **Objectives defined** — Clear targets for each outcome
6. **Error handling in place** — Graceful handling of failed evaluations
7. **Monitoring configured** — Track Pareto growth and metric improvements
8. **Holdout set reserved** — Tasks not used in optimization
9. **Results storage configured** — Where to save optimization artifacts
10. **Cost budget set** — Maximum iterations/evaluations allowed

---

Appendix A: GlobalMOO API reference

Key API endpoints used:

Endpoint	Purpose
`CreateModel`	Define input/output dimensions
`CreateProject`	Set bounds, types, categories
`LoadOutputCases`	Create trial with seed data
`LoadObjectives`	Define optimization targets
`InitializeInverse`	Start inverse optimization
`SuggestInverse`	Get next config to evaluate
`LoadInversedOutput`	Report evaluation results

For full API documentation, see https://globalmoo.gitbook.io/globalmoo-documentation/

Appendix B: Troubleshooting

"Invalid API key"
Check that GLOBALMOO_API_KEY is set correctly and hasn't expired.

"No Pareto frontier found"
Need more seed data or more iterations. Try at least 20 seed cases and 50 iterations.

"All configurations produce similar outcomes"
Your configuration variables might not affect the outcomes much. Check impact factors and consider expanding the search space.

"Optimization not converging"
Check that your objectives are achievable. Relaxing constraints often helps.

"Evaluation failures"
Your forward model is failing. Add logging and retry logic. Check LLM API limits.

End of GlobalMOO Developer Guide