# DSPy Developer Guide

## Programmatic Prompt 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
**DSPy Version:** 2.4+ recommended

---

## 0) What DSPy gives you (and what it doesn't)

DSPy shifts prompt engineering from **string manipulation** to **programmatic optimization**. Instead of crafting prompt text by hand and hoping it works, you define what you want (signatures), compose operations (modules), and let optimizers search for effective expressions.

What DSPy provides:

- Automatic prompt tuning based on measured outcomes
- Few-shot example synthesis from successful runs
- Instruction optimization via Bayesian search
- Modular composition of LLM operations
- Caching and reproducibility infrastructure

What DSPy does NOT provide:

- Multi-objective Pareto optimization (use GlobalMOO for that)
- Framework structure search (that's Phase A / Level 1)
- Automatic task corpus generation
- Magic—you still need good evaluation metrics

In our architecture, DSPy operates at **two levels**:

- **Level 2 (fast, per-cluster):** Optimize prompt expression for a fixed configuration
- **Level 1 (slow, periodic):** Analyze failures and propose structural changes

This guide covers both levels with implementation patterns.

## 1) Core abstractions

### 1.1 Signatures: declaring what you want

A Signature defines the **contract** between you and the LLM. It specifies input fields, output fields, and a docstring that becomes part of the prompt.

```python
import dspy

class AnalyzeWithFrames(dspy.Signature):
    """Analyze a problem using specified cognitive frames and VERIX notation."""

    # Inputs: what the module receives
    task = dspy.InputField(desc="The problem or question to analyze")
    active_frames = dspy.InputField(desc="Cognitive frames to apply (e.g., turkish_evidential, russian_aspect)")
    verix_requirements = dspy.InputField(desc="VERIX notation requirements and compression level")

    # Outputs: what the module produces
    analysis = dspy.OutputField(desc="Structured analysis applying the specified frames")
    confidence_summary = dspy.OutputField(desc="Summary of confidence levels for key claims")
```

The Signature is **declarative**. You specify WHAT, not HOW. DSPy's optimizers figure out how to express this as an effective prompt.

Key design principles for signatures:

- Field descriptions become prompt instructions, so write them carefully
- Keep input/output semantics stable across optimization runs
- Avoid over-specifying format in the signature—let optimizers find what works

### 1.2 Modules: composable operations

A Module wraps one or more Signatures with logic for how to invoke them. Modules can be composed into pipelines.

```python
```

```
class CognitiveAnalyzer(dspy.Module):
    """
    A module that applies VERILINGUA frames and VERIX notation to analysis tasks.
    """

    def __init__(self):
        super().__init__()
        # ChainOfThought adds step-by-step reasoning before the final output
        self.analyze = dspy.ChainOfThought(AnalyzeWithFrames)

    def forward(self, task: str, active_frames: list[str], verix_requirements: str):
        """
        Process a task through frame-activated reasoning.

        Args:
            task: The problem to analyze
            active_frames: List of frame names to activate
            verix_requirements: VERIX notation spec (strictness, compression, etc.)

        Returns:
            DSPy Prediction with 'analysis' and 'confidence_summary' fields
        """
        # Format frames as a readable string for the prompt
        frames_str = ", ".join(active_frames) if active_frames else "none specified"

        # Invoke the signature
        result = self.analyze(
            task=task,
            active_frames=frames_str,
            verix_requirements=verix_requirements
        )

        return result
```

Module composition example—a two-stage pipeline:

```
python
```

```
class AnalyzeThenVerify(dspy.Module):
    """
    First analyze, then verify claims against VERIX consistency requirements.
    """

    def __init__(self):
        super().__init__()
        self.analyzer = CognitiveAnalyzer()
        self.verifier = dspy.ChainOfThought(VerifyConsistency)

    def forward(self, task, active_frames, verix_requirements):
        # Stage 1: Generate analysis
        analysis_result = self.analyzer(task, active_frames, verix_requirements)

        # Stage 2: Verify epistemic consistency
        verification = self.verifier(
            analysis=analysis_result.analysis,
            confidence_summary=analysis_result.confidence_summary
        )

        return dspy.Prediction(
            analysis=analysis_result.analysis,
            confidence_summary=analysis_result.confidence_summary,
            verification=verification.assessment,
            is_consistent=verification.is_consistent
        )
```

### 1.3 Optimizers (Teleprompters): learning what works

Optimizers take a Module and training examples, then search for prompt configurations that maximize a metric. The main optimizers you'll use:

**BootstrapFewShot** synthesizes demonstrations from successful runs. It runs your module on training examples, keeps the ones that pass your metric, and includes them as few-shot examples in future prompts.

python

```
from dspy.teleprompt import BootstrapFewShot

optimizer = BootstrapFewShot(
    metric=your_metric_function,
    max_bootstrapped_demos=4,    # How many examples to include
    max_labeled_demos=4,         # If you have labeled examples too
    max_rounds=1                 # Bootstrap iterations
)

optimized_module = optimizer.compile(
    student=CognitiveAnalyzer(),
    trainset=training_examples
)
```

**MIPROv2** uses Bayesian optimization to search over both instructions and demonstrations. It's more expensive but often finds better configurations.

```python
from dspy.teleprompt import MIPROv2

optimizer = MIPROv2(
    metric=your_metric_function,
    auto="medium",  # "light", "medium", or "heavy" search intensity
    num_candidates=10,
    init_temperature=1.0
)

optimized_module = optimizer.compile(
    student=CognitiveAnalyzer(),
    trainset=training_examples,
    eval_kwargs={"num_threads": 4}  # Parallel evaluation
)
```

**GEPA** (Grounded Example-based Prompt Adaptation) uses the LLM to critique and improve its own prompts. Useful for periodic refinement.

```python
```

```
from dspy.teleprompt import GEPA

optimizer = GEPA(
    metric=your_metric_function,
    max_iterations=5
)

optimized_module = optimizer.compile(
    student=CognitiveAnalyzer(),
    trainset=training_examples
)
```

### 1.4 Metrics: what "good" means

A metric function evaluates a prediction against expected outcomes. It returns a score (typically 0-1 or boolean).

```
python
```

```python
def cognitive_quality_metric(example, prediction, trace=None) -> float:
    """
    Evaluate a cognitive analysis prediction.

    Args:
        example: The input example (has 'task', expected 'reference' if available)
        prediction: The module's output (has 'analysis', 'confidence_summary')
        trace: Optional execution trace for debugging

    Returns:
        Score from 0.0 to 1.0
    """
    score = 0.0

    # 1) Check VERIX format compliance
    format_score = verix_parser.format_compliance(prediction.analysis)
    score += format_score * 0.3

    # 2) Check frame activation compliance
    frame_markers_present = check_frame_markers(prediction.analysis, example.active_frames)
    score += frame_markers_present * 0.2

    # 3) If we have a reference, check accuracy
    if hasattr(example, 'reference') and example.reference:
        accuracy = compute_accuracy(prediction.analysis, example.reference)
        score += accuracy * 0.3
    else:
        # For open-ended tasks, reward structure over emptiness
        score += min(1.0, len(prediction.analysis) / 500) * 0.3

    # 4) Confidence extraction and consistency
    claims = verix_parser.parse(prediction.analysis)
    if claims:
        consistency = epistemic_consistency(claims)
        score += consistency * 0.2
    else:
        score += 0.1  # Partial credit for producing output

    return score
```

## 2) Level 2: Prompt expression optimization

Level 2 is the **inner loop** of our architecture. Given a fixed configuration (which frames are active, what VERIX strictness, etc.), Level 2 finds the best way to express that configuration as prompts.

### 2.1 The clustering strategy

You don't want to run DSPy optimization for every single configuration GlobalMOO tries. That would be prohibitively expensive. Instead, cluster similar configurations and share optimized prompts within each cluster.

Configurations belong to the same cluster if they share:

- The same active frame set (which frames are enabled)
- The same VERIX strictness level
- The same activation style (implicit/explicit/checklist)

Configurations can differ in:

- max_frames_per_task (affects selection, not prompt structure)
- fewshot_count (affects length, not core structure)
- verbosity level (affects detail, minor structural impact)

python

```python
import hashlib
from dataclasses import dataclass
from typing import Optional

@dataclass
class PromptCluster:
    """
    A cluster of configurations that share the same prompt structure.
    """
    cluster_id: str
    signature_hash: str

    # The optimized module for this cluster (None until compiled)
    compiled_module: Optional[dspy.Module] = None

    # Performance stats for monitoring
    compilation_date: Optional[str] = None
    training_examples_used: int = 0
    validation_score: float = 0.0


def compute_cluster_signature(config: FullConfig) -> str:
    """
    Compute a stable hash that identifies which cluster a config belongs to.

    Configs with the same signature share a compiled module.
    """
    # Extract the structural components that define the cluster
    components = [
        # Frame enables (sorted for stability)
        str(sorted([
            f for f in [
                'russian_aspect' if config.framework.russian_aspect else None,
                'turkish_evidential' if config.framework.turkish_evidential else None,
                'arabic_morphology' if config.framework.arabic_morphology else None,
                'german_composition' if config.framework.german_composition else None,
                'japanese_keigo' if config.framework.japanese_keigo else None,
                'chinese_classifiers' if config.framework.chinese_classifiers else None,
                'guugu_spatial' if config.framework.guugu_spatial else None,
            ] if f is not None
        ])),
        # VERIX strictness
        str(config.framework.verix_strictness),
        # Activation style
        str(config.prompt.activation_style),
    ]

    signature_string = "|".join(components)
    return hashlib.md5(signature_string.encode()).hexdigest()[:12]
```

### 2.2 The PromptClusterManager

This class manages compilation and caching of optimized modules per cluster.

```
python
```

```python
import json
from pathlib import Path
from datetime import datetime
import dspy

class PromptClusterManager:
    """
    Manages DSPy-compiled modules for configuration clusters.

    Responsibilities:
    - Track which clusters exist
    - Compile new clusters when needed
    - Cache and load compiled modules
    - Provide the right module for any configuration
    """

    def __init__(
        self,
        cache_dir: str = "./storage/prompts",
        lm: Optional[dspy.LM] = None
    ):
        self.cache_dir = Path(cache_dir)
        self.cache_dir.mkdir(parents=True, exist_ok=True)

        self.clusters: dict[str, PromptCluster] = {}
        self.lm = lm or dspy.Claude(model="claude-sonnet-4-20250514")

        # Load existing clusters from cache
        self._load_cluster_index()

    def get_module_for_config(self, config: FullConfig) -> dspy.Module:
        """
        Get the appropriate compiled module for a configuration.

        If the cluster hasn't been compiled yet, returns an uncompiled
        base module (will work, just not optimized).
        """
        signature = compute_cluster_signature(config)

        if signature in self.clusters:
            cluster = self.clusters[signature]
            if cluster.compiled_module is not None:
                return cluster.compiled_module
    # No compiled module available—return base module
    return self._create_base_module(config)

def compile_cluster(
    self,
    config: FullConfig,
    training_examples: list,
    metric: callable,
    optimizer_type: str = "mipro"
) -> PromptCluster:
    """
    Compile an optimized module for a configuration's cluster.

    Args:
        config: Any config from the target cluster
        training_examples: Examples for optimization
        metric: Evaluation metric function
        optimizer_type: "bootstrap", "mipro", or "gepa"

    Returns:
        The updated PromptCluster
    """
    signature = compute_cluster_signature(config)

    # Create base module
    base_module = self._create_base_module(config)

    # Select optimizer
    if optimizer_type == "bootstrap":
        optimizer = dspy.teleprompt.BootstrapFewShot(
            metric=metric,
            max_bootstrapped_demos=4
        )
    elif optimizer_type == "mipro":
        optimizer = dspy.teleprompt.MIPROv2(
            metric=metric,
            auto="medium"
        )
    elif optimizer_type == "gepa":
        optimizer = dspy.teleprompt.GEPA(
            metric=metric,
            max_iterations=3
        )
        else:
            raise ValueError(f"Unknown optimizer: {optimizer_type}")

        # Configure DSPy with the language model
        with dspy.context(lm=self.lm):
            compiled_module = optimizer.compile(
                student=base_module,
                trainset=training_examples
            )

        # Create or update cluster
        cluster = PromptCluster(
            cluster_id=signature,
            signature_hash=signature,
            compiled_module=compiled_module,
            compilation_date=datetime.now().isoformat(),
            training_examples_used=len(training_examples),
            validation_score=self._evaluate_module(compiled_module, training_examples, metric)
        )

        self.clusters[signature] = cluster

        # Save to cache
        self._save_cluster(cluster)
        self._save_cluster_index()

        return cluster

    def _create_base_module(self, config: FullConfig) -> dspy.Module:
        """
        Create an uncompiled module configured for the given config.
        """
        # Build the VERIX requirements string based on config
        verix_req = self._build_verix_requirements(config)

        # Get active frames
        active_frames = self._get_active_frames(config)

        # Create a module that captures these settings
        class ConfiguredAnalyzer(dspy.Module):
            def __init__(self, frames, verix):
                super().__init__()
                self.frames = frames
                self.verix = verix
                self.analyze = dspy.ChainOfThought(AnalyzeWithFrames)

            def forward(self, task):
                return self.analyze(
                    task=task,
                    active_frames=", ".join(self.frames),
                    verix_requirements=self.verix
                )

        return ConfiguredAnalyzer(active_frames, verix_req)

    def _build_verix_requirements(self, config: FullConfig) -> str:
        """Build VERIX requirements string from config."""
        parts = []

        strictness_map = {0: "optional", 1: "partial", 2: "full"}
        parts.append(f"VERIX strictness: {strictness_map[config.framework.verix_strictness]}")

        if config.framework.verix_confidence_required:
            parts.append("Confidence levels required for all claims")
        if config.framework.verix_ground_chain_required:
            parts.append("Ground chains required for factual claims")

        return "; ".join(parts)

    def _get_active_frames(self, config: FullConfig) -> list[str]:
        """Extract list of active frame names from config."""
        frames = []
        if config.framework.russian_aspect:
            frames.append("russian_aspect")
        if config.framework.turkish_evidential:
            frames.append("turkish_evidential")
        if config.framework.arabic_morphology:
            frames.append("arabic_morphology")
        if config.framework.german_composition:
            frames.append("german_composition")
        if config.framework.japanese_keigo:
            frames.append("japanese_keigo")
        if config.framework.chinese_classifiers:
            frames.append("chinese_classifiers")
        if config.framework.guugu_spatial:
            frames.append("guugu_spatial")
        return frames
```

```
def _evaluate_module(self, module, examples, metric) -> float:
    """Evaluate a module on examples and return average score."""
    scores = []
    for ex in examples:
        try:
            pred = module(task=ex.task)
            score = metric(ex, pred)
            scores.append(score)
        except Exception:
            scores.append(0.0)
    return sum(scores) / len(scores) if scores else 0.0

def _save_cluster(self, cluster: PromptCluster):
    """Save a compiled cluster to disk."""
    cluster_path = self.cache_dir / f"cluster_{cluster.cluster_id}.json"

    # DSPy modules can be saved via their state
    if cluster.compiled_module:
        module_state = cluster.compiled_module.dump_state()
    else:
        module_state = None

    data = {
        "cluster_id": cluster.cluster_id,
        "signature_hash": cluster.signature_hash,
        "compilation_date": cluster.compilation_date,
        "training_examples_used": cluster.training_examples_used,
        "validation_score": cluster.validation_score,
        "module_state": module_state
    }

    with open(cluster_path, 'w') as f:
        json.dump(data, f, indent=2)

def _load_cluster_index(self):
    """Load the cluster index from disk."""
    index_path = self.cache_dir / "cluster_index.json"
    if index_path.exists():
        with open(index_path) as f:
            index = json.load(f)

        for cluster_id in index.get("clusters", []):
            self._load_cluster(cluster_id)
    def _load_cluster(self, cluster_id: str):
        """Load a single cluster from disk."""
        cluster_path = self.cache_dir / f"cluster_{cluster_id}.json"
        if not cluster_path.exists():
            return

        with open(cluster_path) as f:
            data = json.load(f)

        cluster = PromptCluster(
            cluster_id=data["cluster_id"],
            signature_hash=data["signature_hash"],
            compilation_date=data.get("compilation_date"),
            training_examples_used=data.get("training_examples_used", 0),
            validation_score=data.get("validation_score", 0.0)
        )

        # Restore module from state if available
        if data.get("module_state"):
            # Create a base module and load state
            # (This requires knowing the config, which we'd need to store too)
            # For now, leave as None—will recompile on first use
            pass

        self.clusters[cluster_id] = cluster

    def _save_cluster_index(self):
        """Save the cluster index to disk."""
        index_path = self.cache_dir / "cluster_index.json"
        index = {
            "clusters": list(self.clusters.keys()),
            "last_updated": datetime.now().isoformat()
        }
        with open(index_path, 'w') as f:
            json.dump(index, f, indent=2)
```

### 2.3 Integration with GlobalMOO evaluation

When GlobalMOO requests evaluation of a configuration, use the cluster manager to get the appropriate optimized module:

python

```
class EvaluationEngine:
    """
    Runs evaluations for GlobalMOO using DSPy-optimized modules.
    """

    def __init__(
        self,
        cluster_manager: PromptClusterManager,
        task_corpus: list,
        lm: dspy.LM
    ):
        self.cluster_manager = cluster_manager
        self.task_corpus = task_corpus
        self.lm = lm

    def evaluate_config(self, config: FullConfig) -> dict[str, float]:
        """
        Evaluate a configuration and return outcome metrics.

        This is the function GlobalMOO calls in the optimization loop.
        """
        # Get the appropriate module for this config's cluster
        module = self.cluster_manager.get_module_for_config(config)

        outcomes = {
            "task_accuracy": [],
            "calibration_score": [],
            "format_compliance": [],
            "tokens_total": [],
            "latency_seconds": [],
            "edge_case_robustness": [],
            "epistemic_consistency": []
        }

        with dspy.context(lm=self.lm):
            for task in self.task_corpus:
                result = self._evaluate_single_task(module, task, config)
                for key, value in result.items():
                    outcomes[key].append(value)

        # Aggregate (mean)
        return {k: sum(v) / len(v) for k, v in outcomes.items()}
    def _evaluate_single_task(self, module, task, config) -> dict[str, float]:
        """Evaluate a single task and measure all outcomes."""
        import time

        start = time.time()

        try:
            prediction = module(task=task.prompt)
            latency = time.time() - start

            # Measure outcomes
            return {
                "task_accuracy": self._measure_accuracy(prediction, task),
                "calibration_score": self._measure_calibration(prediction, task),
                "format_compliance": self._measure_format(prediction, config),
                "tokens_total": self._count_tokens(prediction),
                "latency_seconds": latency,
                "edge_case_robustness": self._measure_robustness(prediction, task),
                "epistemic_consistency": self._measure_consistency(prediction)
            }
        except Exception as e:
            # On failure, return worst-case metrics
            return {
                "task_accuracy": 0.0,
                "calibration_score": 1.0,  # Worst (Brier)
                "format_compliance": 0.0,
                "tokens_total": 10000,  # Penalty
                "latency_seconds": 30.0,  # Timeout assumption
                "edge_case_robustness": 0.0,
                "epistemic_consistency": 0.0
            }
```

---

## 3) Level 1: Framework evolution

Level 1 operates at a slower timescale—monthly rather than per-evaluation. It analyzes accumulated logs to propose structural changes to VERILINGUA and VERIX.

### 3.1 What Level 1 does

Level 1 answers questions like:

- Which frames are actually contributing value? (Impact analysis)

- What failure patterns keep recurring? (Failure taxonomy)
- Should we add a new frame or remove an underperforming one? (Structural proposals)
- Are routing rules working, or do certain task types need different frame combinations? (Routing refinement)

### 3.2 Failure taxonomy

First, classify failures into actionable categories:

python

```python
from dataclasses import dataclass
from enum import Enum
from typing import Optional

class FailureType(Enum):
    # Epistemic failures
    OVERCONFIDENCE = "overconfidence"       # High stated confidence, wrong answer
    UNDERCONFIDENCE = "underconfidence"     # Low stated confidence, right answer
    GROUNDING_FAILURE = "grounding_failure" # Claims without proper sourcing
    INCONSISTENCY = "inconsistency"         # Contradictory confidence levels

    # Structural failures
    FRAME_IGNORED = "frame_ignored"         # Frame activated but not applied
    VERIX_VIOLATION = "verix_violation"     # VERIX format not followed
    FOCUS_DRIFT = "focus_drift"             # Response wandered off-task

    # Performance failures
    VERBOSITY = "verbosity"                 # Excessive length
    TERSENESS = "terseness"                 # Insufficient detail
    LATENCY = "latency"                     # Too slow

    # Domain failures
    MATH_ERROR = "math_error"               # Calculation mistake
    LOGIC_ERROR = "logic_error"             # Reasoning flaw
    FACTUAL_ERROR = "factual_error"         # Wrong fact


@dataclass
class FailureCase:
    """A recorded failure for analysis."""
    config_vector: list[int]
    task_id: str
    task_type: str
    failure_type: FailureType
    severity: float  # 0-1, how bad
    response_excerpt: str
    analysis: str


class FailureClassifier:
    """
    Classifies evaluation outcomes into failure types.
    """
```

```
def classify(
    self,
    task,
    prediction,
    outcomes: dict[str, float],
    config: FullConfig
) -> Optional[FailureCase]:
    """
    Analyze an outcome and classify any failure.

    Returns None if no significant failure detected.
    """
    failures = []

    # Check for overconfidence
    if outcomes["calibration_score"] > 0.25:  # Brier score threshold
        claims = verix_parser.parse(prediction.analysis)
        high_conf_claims = [c for c in claims if c.confidence and c.confidence > 0.8]
        if high_conf_claims and outcomes["task_accuracy"] < 0.5:
            failures.append((FailureType.OVERCONFIDENCE, 0.8))

    # Check for grounding failure
    if config.framework.verix_ground_chain_required:
        if outcomes["format_compliance"] < 0.5:
            failures.append((FailureType.GROUNDING_FAILURE, 0.6))

    # Check for frame being ignored
    active_frames = self._get_active_frames(config)
    for frame in active_frames:
        if not self._frame_markers_present(prediction.analysis, frame):
            failures.append((FailureType.FRAME_IGNORED, 0.5))
            break

    # Check for verbosity
    if outcomes["tokens_total"] > 2000 and outcomes["task_accuracy"] < 0.7:
        failures.append((FailureType.VERBOSITY, 0.4))

    # Check for focus drift
    if outcomes.get("focus_score", 1.0) < 0.5:
        failures.append((FailureType.FOCUS_DRIFT, 0.6))

    # Return most severe failure
    if failures:
            failures.sort(key=lambda x: -x[1])
            failure_type, severity = failures[0]

            return FailureCase(
                config_vector=VectorCodec().to_vector(config),
                task_id=task.id,
                task_type=task.category,
                failure_type=failure_type,
                severity=severity,
                response_excerpt=prediction.analysis[:500],
                analysis=self._generate_analysis(failure_type, config, prediction)
            )

        return None

    def _generate_analysis(self, failure_type, config, prediction) -> str:
        """Generate human-readable analysis of the failure."""
        analyses = {
            FailureType.OVERCONFIDENCE: (
                f"High confidence claims made despite low accuracy. "
                f"VERIX strictness was {config.framework.verix_strictness}. "
                f"Consider requiring confidence calibration training."
            ),
            FailureType.GROUNDING_FAILURE: (
                f"Ground chains required but not provided. "
                f"Turkish evidential frame was {'active' if config.framework.turkish_evidential else 'inactive'}. "
                f"Consider stricter frame enforcement."
            ),
            FailureType.FRAME_IGNORED: (
                f"Frame activation instructions present but frame distinctions not made in output. "
                f"Activation style was {config.prompt.activation_style}. "
                f"Consider checklist activation or more explicit instructions."
            ),
            FailureType.VERBOSITY: (
                f"Excessive token usage without proportional accuracy. "
                f"System prompt length was {config.prompt.system_prompt_length}. "
                f"Consider L0/L1 compression or brevity instructions."
            ),
        }
        return analyses.get(failure_type, "Unanalyzed failure type")
```

### 3.3 Impact analysis

Measure which configuration variables actually affect which outcomes:

```
python
```

```
import numpy as np
from typing import Dict, List, Tuple

class ImpactAnalyzer:
    """
    Analyzes which configuration variables impact which outcomes.

    This helps identify:
    - High-impact variables (tune these carefully)
    - Low-impact variables (candidates for removal)
    - Interaction effects (variables that matter together)
    """

    def __init__(self, logs: List[dict]):
        """
        Args:
            logs: List of evaluation logs, each with 'config_vector' and 'outcomes'
        """
        self.logs = logs
        self.config_matrix = np.array([log['config_vector'] for log in logs])
        self.outcome_matrix = np.array([
            [log['outcomes'][k] for k in sorted(log['outcomes'].keys())]
            for log in logs
        ])
        self.outcome_names = sorted(logs[0]['outcomes'].keys())

    def compute_correlations(self) -> Dict[str, Dict[str, float]]:
        """
        Compute correlation between each config variable and each outcome.

        Returns:
            Nested dict: variable_name -> outcome_name -> correlation
        """
        variable_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"
        ]

        correlations = {}
        for i, var_name in enumerate(variable_names):
            var_values = self.config_matrix[:, i]
            correlations[var_name] = {}

            for j, outcome_name in enumerate(self.outcome_names):
                outcome_values = self.outcome_matrix[:, j]

                # Compute Pearson correlation
                corr = np.corrcoef(var_values, outcome_values)[0, 1]
                correlations[var_name][outcome_name] = float(corr) if np.isfinite(corr) else 0.0

        return correlations

    def identify_high_impact_variables(self, threshold: float = 0.3) -> List[Tuple[str, str, float]]:
        """
        Find variable-outcome pairs with strong correlation.

        Returns:
            List of (variable, outcome, correlation) tuples
        """
        correlations = self.compute_correlations()
        high_impact = []

        for var_name, outcome_corrs in correlations.items():
            for outcome_name, corr in outcome_corrs.items():
                if abs(corr) >= threshold:
                    high_impact.append((var_name, outcome_name, corr))

        # Sort by absolute correlation
        high_impact.sort(key=lambda x: -abs(x[2]))
        return high_impact

    def identify_low_impact_variables(self, threshold: float = 0.05) -> List[str]:
        """
        Find variables with low impact on all outcomes.

        These are candidates for removal from the search space.
        """
        correlations = self.compute_correlations()
        low_impact = []

        for var_name, outcome_corrs in correlations.items():
            max_impact = max(abs(c) for c in outcome_corrs.values())
            if max_impact < threshold:
                low_impact.append(var_name)

        return low_impact

    def ablation_analysis(self, variable_name: str) -> Dict[str, Tuple[float, float]]:
        """
        Compare outcomes when a variable is on vs off.

        Returns:
            outcome_name -> (mean_when_off, mean_when_on)
        """
        var_idx = [
            "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"
        ].index(variable_name)

        var_values = self.config_matrix[:, var_idx]

        # For boolean variables, split on 0/1
        # For others, split on median
        if set(var_values).issubset({0, 1}):
            off_mask = var_values == 0
            on_mask = var_values == 1
        else:
            median = np.median(var_values)
            off_mask = var_values <= median
            on_mask = var_values > median

        results = {}
        for j, outcome_name in enumerate(self.outcome_names):
            outcome_values = self.outcome_matrix[:, j]
            mean_off = np.mean(outcome_values[off_mask]) if any(off_mask) else 0
            mean_on = np.mean(outcome_values[on_mask]) if any(on_mask) else 0
            results[outcome_name] = (float(mean_off), float(mean_on))

        return results
```

### 3.4 Proposal generation

Based on failure analysis and impact analysis, generate structural proposals:

python

```
@dataclass
class FrameworkProposal:
    """A proposed change to the framework structure."""
    proposal_type: str  # "add_frame", "remove_frame", "modify_routing", "adjust_verix"
    description: str
    rationale: str
    evidence: dict
    estimated_impact: dict[str, float]
    priority: int  # 1=high, 2=medium, 3=low


class ProposalGenerator:
    """
    Generates framework evolution proposals based on analysis.
    """

    def __init__(
        self,
        failure_cases: List[FailureCase],
        impact_analysis: ImpactAnalyzer
    ):
        self.failures = failure_cases
        self.impact = impact_analysis

    def generate_proposals(self) -> List[FrameworkProposal]:
        """Generate all proposals based on current evidence."""
        proposals = []

        # Analyze failure patterns
        proposals.extend(self._proposals_from_failures())

        # Analyze impact patterns
        proposals.extend(self._proposals_from_impact())

        # Sort by priority
        proposals.sort(key=lambda p: p.priority)

        return proposals

    def _proposals_from_failures(self) -> List[FrameworkProposal]:
        """Generate proposals based on failure patterns."""
        proposals = []
        # Count failure types
        failure_counts = {}
        for fc in self.failures:
            ft = fc.failure_type
            failure_counts[ft] = failure_counts.get(ft, 0) + 1

        # Propose fixes for common failures
        if failure_counts.get(FailureType.GROUNDING_FAILURE, 0) > 5:
            proposals.append(FrameworkProposal(
                proposal_type="adjust_verix",
                description="Make ground chain requirement default to True",
                rationale=f"{failure_counts[FailureType.GROUNDING_FAILURE]} grounding failures observed",
                evidence={"failure_count": failure_counts[FailureType.GROUNDING_FAILURE]},
                estimated_impact={"format_compliance": 0.2, "tokens_total": 50},
                priority=1
            ))

        if failure_counts.get(FailureType.OVERCONFIDENCE, 0) > 5:
            proposals.append(FrameworkProposal(
                proposal_type="modify_routing",
                description="Enable Turkish evidential frame by default for sourcing tasks",
                rationale=f"{failure_counts[FailureType.OVERCONFIDENCE]} overconfidence failures observed",
                evidence={"failure_count": failure_counts[FailureType.OVERCONFIDENCE]},
                estimated_impact={"calibration_score": -0.1, "tokens_total": 30},
                priority=1
            ))

        if failure_counts.get(FailureType.FRAME_IGNORED, 0) > 3:
            proposals.append(FrameworkProposal(
                proposal_type="modify_routing",
                description="Switch default activation style from implicit to explicit",
                rationale=f"{failure_counts[FailureType.FRAME_IGNORED]} cases of frame instructions being ignored",
                evidence={"failure_count": failure_counts[FailureType.FRAME_IGNORED]},
                estimated_impact={"format_compliance": 0.15},
                priority=2
            ))

        return proposals

    def _proposals_from_impact(self) -> List[FrameworkProposal]:
        """Generate proposals based on impact analysis."""
        proposals = []

        # Check for low-impact variables
        low_impact_vars = self.impact.identify_low_impact_variables(threshold=0.05)

        for var in low_impact_vars:
            if var in ["japanese_keigo", "chinese_classifiers", "guugu_spatial"]:
                proposals.append(FrameworkProposal(
                    proposal_type="remove_frame",
                    description=f"Consider removing {var} frame from search space",
                    rationale=f"Variable has <5% impact on all measured outcomes",
                    evidence={"correlations": self.impact.compute_correlations()[var]},
                    estimated_impact={"search_efficiency": 0.1},
                    priority=3
                ))

        # Check for high-impact variables that might need finer control
        high_impact = self.impact.identify_high_impact_variables(threshold=0.4)

        for var, outcome, corr in high_impact:
            if var == "verix_strictness" and outcome == "calibration_score":
                proposals.append(FrameworkProposal(
                    proposal_type="adjust_verix",
                    description="VERIX strictness has strong calibration impact—consider task-dependent defaults",
                    rationale=f"Correlation of {corr:.2f} between strictness and calibration",
                    evidence={"correlation": corr},
                    estimated_impact={},
                    priority=2
                ))

        return proposals
```

### 3.5 Running Level 1 analysis

Bring it together in a monthly analysis workflow:

```
python
```

```python
def run_level1_analysis(
    logs_dir: str,
    output_dir: str,
    min_logs: int = 100
) -> dict:
    """
    Run monthly Level 1 framework evolution analysis.

    Args:
        logs_dir: Directory containing JSONL evaluation logs
        output_dir: Where to write analysis results
        min_logs: Minimum logs required for analysis

    Returns:
        Analysis results including proposals
    """
    # Load logs
    logs = load_logs_from_jsonl(logs_dir)

    if len(logs) < min_logs:
        return {"status": "insufficient_data", "logs_found": len(logs)}

    # Classify failures
    classifier = FailureClassifier()
    failures = []

    for log in logs:
        config = VectorCodec().from_vector(log['config_vector'])
        task = Task(id=log['task_id'], category=log['task_type'], prompt=log['task_prompt'])
        prediction = type('Prediction', (), {'analysis': log['response']})()

        failure = classifier.classify(task, prediction, log['outcomes'], config)
        if failure:
            failures.append(failure)

    # Run impact analysis
    impact = ImpactAnalyzer(logs)

    # Generate proposals
    generator = ProposalGenerator(failures, impact)
    proposals = generator.generate_proposals()

    # Compile results
    results = {
        "status": "complete",
        "logs_analyzed": len(logs),
        "failures_identified": len(failures),
        "failure_breakdown": {
            ft.value: len([f for f in failures if f.failure_type == ft])
            for ft in FailureType
        },
        "high_impact_variables": impact.identify_high_impact_variables(),
        "low_impact_variables": impact.identify_low_impact_variables(),
        "proposals": [
            {
                "type": p.proposal_type,
                "description": p.description,
                "rationale": p.rationale,
                "priority": p.priority
            }
            for p in proposals
        ]
    }

    # Save results
    output_path = Path(output_dir) / f"level1_analysis_{datetime.now().strftime('%Y%m')}.json"
    with open(output_path, 'w') as f:
        json.dump(results, f, indent=2)

    return results
```

---

## 4) Practical patterns and common issues

### 4.1 Handling DSPy's token costs

DSPy optimization requires many LLM calls. Manage costs by:

```python
```

```
# Use smaller/cheaper models for optimization, deploy with larger
optimization_lm = dspy.Claude(model="claude-haiku-4-20250514")  # Cheaper
production_lm = dspy.Claude(model="claude-sonnet-4-20250514")   # Better

# Compile with cheap model
with dspy.context(lm=optimization_lm):
    compiled = optimizer.compile(module, trainset)

# Deploy with production model
with dspy.context(lm=production_lm):
    result = compiled(task=user_task)
```

### 4.2 Caching to avoid recompilation

DSPy provides caching, but manage it explicitly for reproducibility:

```python
import dspy

# Enable DSPy's built-in caching
dspy.settings.configure(
    cache_turn_on=True,
    cache_seed=42  # For reproducibility
)

# Or use your own caching layer
from functools import lru_cache

@lru_cache(maxsize=1000)
def cached_module_call(task_hash: str, cluster_id: str):
    module = cluster_manager.get_module_for_config(...)
    return module(task=...)
```

### 4.3 Debugging optimization failures

When optimization doesn't improve metrics:

```python
```

```
# Enable verbose logging
import logging
logging.getLogger("dspy").setLevel(logging.DEBUG)

# Check what the optimizer is trying
optimizer = MIPROv2(
    metric=your_metric,
    auto="medium",
    verbose=True  # Print optimization progress
)

# Inspect the compiled module's prompts
compiled = optimizer.compile(module, trainset)
print(compiled.analyze.extended_signature)  # See the learned prompt
```

### 4.4 Metric design pitfalls

Common metric issues and fixes:

```python
# BAD: Metric that can be gamed by verbosity
def bad_metric(example, pred):
    return len(pred.analysis) / 1000  # Longer = better??

# GOOD: Metric with length normalization
def good_metric(example, pred):
    accuracy = compute_accuracy(pred, example)
    length_penalty = max(0, 1 - (len(pred.analysis) - 500) / 1000)
    return accuracy * 0.7 + length_penalty * 0.3

# BAD: Binary metric (no gradient signal)
def bad_metric(example, pred):
    return 1.0 if "correct" in pred.analysis else 0.0

# GOOD: Continuous metric with partial credit
def good_metric(example, pred):
    partial_scores = []
    for criterion in evaluation_criteria:
        partial_scores.append(criterion.score(pred))
    return sum(partial_scores) / len(partial_scores)
```

## 5) Integration checklist

Before deploying DSPy integration:

1. **Signatures defined** — Input/output contracts clear and stable
2. **Modules composed** — Pipeline structure decided
3. **Metric implemented** — Evaluation function with good gradient signal
4. **Clustering implemented** — Signature hashing and cluster management
5. **Cache configured** — Persistent storage for compiled modules
6. **Failure classifier ready** — For Level 1 analysis
7. **Logging schema defined** — Captures config, outcomes, and response excerpts
8. **Holdout set reserved** — Examples optimizers never see
9. **Cost budget set** — Token limits for optimization runs
10. **Monitoring in place** — Track optimization improvements over time

---

## Appendix A: DSPy version compatibility

This guide targets DSPy 2.4+. Key changes from earlier versions:

- `Teleprompter` renamed to optimizer classes in `dspy.teleprompt`
- `dspy.settings.configure()` for global settings
- `module.dump_state()` / `module.load_state()` for serialization
- `dspy.context(lm=...)` for scoped LM configuration

Check https://dspy.ai for current documentation.

---

## Appendix B: Example training data format

```
python
```

```python
from dspy import Example

# Create training examples
trainset = [
    Example(
        task="Analyze the trade-offs between solar and wind energy for urban deployment",
        active_frames=["turkish_evidential", "german_composition"],
        verix_requirements="VERIX partial; confidence required",
        # Optional: expected output for supervised training
        reference_analysis="..."
    ).with_inputs("task", "active_frames", "verix_requirements"),

    Example(
        task="Calculate the compound interest on $10,000 at 5% over 10 years",
        active_frames=["russian_aspect"],
        verix_requirements="VERIX full; ground chain required"
    ).with_inputs("task", "active_frames", "verix_requirements"),

    # ... more examples
]
```

---

**End of DSPy Developer Guide**