# Untitled

# GEPA Universal Prompt Optimizer


> **A production-ready Python library for universal prompt optimization using the GEPA (Generative Evaluation and Prompt Adaptation) framework. Built for developers who need reliable, scalable prompt optimization with comprehensive evaluation metrics.**

## 🎯 What is GEPA Optimizer?

GEPA Optimizer is a sophisticated framework that automatically improves prompts for Large Language Models (LLMs) by iteratively evaluating and refining them based on performance metrics. Think of it as "AutoML for prompts" - it takes your initial prompt and dataset, then uses advanced optimization techniques to create a better-performing prompt.

### Key Capabilities

- 🔄 **Universal Prompt Optimization**: Works with any LLM provider (OpenAI, Anthropic, Google, Hugging Face)
- 👁️ **Multi-Modal Support**: Optimize prompts for vision-capable models (GPT-4V, Claude-3, Gemini)
- 📊 **Advanced Evaluation**: Comprehensive metrics for UI tree extraction and general prompt performance
- 🏭 **Production Ready**: Enterprise-grade reliability with async support, error handling, and monitoring
- ⚙️ **Flexible Configuration**: Easy-to-use configuration system for any optimization scenario
- 💰 **Cost Optimization**: Built-in budget controls and cost estimation
- 🎨 **UI Tree Extraction**: Specialized for optimizing UI interaction and screen understanding tasks
- 🔧 **Extensible Architecture**: Create custom evaluators and adapters for any use case

# 🚀 Quick Start

## Installation

```
pip install gepa-optimizer
```

## Basic Usage

```python
import asyncio
from gepa_optimizer import GepaOptimizer, OptimizationConfig

async def optimize_prompt():
    # Configure your optimization
    config = OptimizationConfig(
        model="openai/gpt-4o",                    # Your target model
        reflection_model="openai/gpt-4o",         # Model for self-
reflection
        max_iterations=10,                        # Budget:
optimization rounds
        max_metric_calls=50,                      # Budget: evaluation
calls
        batch_size=4                              # Batch size for
evaluation
    )

    # Initialize optimizer
    optimizer = GepaOptimizer(config=config)

    # Your training data
    dataset = [
        {"input": "What is artificial intelligence?", "output": "AI is
a field of computer science..."},
        {"input": "Explain machine learning", "output": "Machine
learning is a subset of AI..."},
        {"input": "What are neural networks?", "output": "Neural
networks are computing systems..."}
    ]

    # Optimize your prompt
    result = await optimizer.train(
        seed_prompt="You are a helpful AI assistant that explains
technical concepts clearly.",
        dataset=dataset
    )

    print(f"✅ Optimization completed!")
    print(f"📈 Performance improvement:
{result.improvement_percent:.2f}%")
    print(f"🎯 Optimized prompt: {result.prompt}")
    print(f"⏱️ Time taken: {result.optimization_time:.2f}s")

# Run the optimization
asyncio.run(optimize_prompt())
```

## UI Tree Extraction (Vision Models)

```
import asyncio
from gepa_optimizer import GepaOptimizer, OptimizationConfig

async def optimize_ui_prompt():
    config = OptimizationConfig(
        model="openai/gpt-4o",                    # Vision-capable
model
        reflection_model="openai/gpt-4o",
        max_iterations=15,
        max_metric_calls=75,
        batch_size=3
    )

    optimizer = GepaOptimizer(config=config)

    # UI screenshot data with images and JSON trees
    dataset = {
        'json_dir': 'json_tree',           # Directory with ground
truth JSON files
        'screenshots_dir': 'screenshots',  # Directory with UI
screenshots
        'type': 'ui_tree_dataset'          # Dataset type
    }

    result = await optimizer.train(
        seed_prompt="Analyze the UI screenshot and extract all elements
as JSON.",
        dataset=dataset
    )

    print(f"🎯 Optimized UI prompt: {result.prompt}")
    print(f"📊 Improvement: {result.improvement_percent:.2f}%")

asyncio.run(optimize_ui_prompt())
```

## Universal Custom Use Cases

```python
import asyncio
from gepa_optimizer import (
    GepaOptimizer, OptimizationConfig, ModelConfig,
    BaseEvaluator, BaseLLMClient, VisionLLMClient
)

class CustomEvaluator(BaseEvaluator):
    """Your custom evaluation logic for any use case"""

    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        # Implement your custom metrics
        accuracy = calculate_accuracy(predicted, expected)
        relevance = calculate_relevance(predicted, expected)

        return {
            "accuracy": accuracy,
            "relevance": relevance,
            "composite_score": (accuracy + relevance) / 2
        }

async def custom_optimization():
    # Create your custom components
    llm_client = VisionLLMClient(
        provider="openai",
        model_name="gpt-4o",
        api_key="your-api-key"
    )
    evaluator = CustomEvaluator()

    # Configure optimization
    config = OptimizationConfig(
        model="openai/gpt-4o",
        reflection_model="openai/gpt-4o",
        max_iterations=10,
        max_metric_calls=50
    )

    # Use universal adapter
    optimizer = GepaOptimizer(
        config=config,
        adapter_type="universal",
        llm_client=llm_client,
        evaluator=evaluator
    )

    # Your custom dataset
    dataset = [
```

```
        {"input": "Your input", "output": "Expected output"},
        # ... more examples
    ]

    result = await optimizer.train(
        seed_prompt="Your initial prompt",
        dataset=dataset
    )

    print(f"🎯 Optimized prompt: {result.prompt}")

asyncio.run(custom_optimization())
```

# 📚 Comprehensive Examples

## 1. Multi-Provider Setup

```
from gepa_optimizer import OptimizationConfig, ModelConfig

# Using different providers for main and reflection models
config = OptimizationConfig(
    model=ModelConfig(
        provider="openai",
        model_name="gpt-4o",
        api_key="your-openai-key",
        temperature=0.7
    ),
    reflection_model=ModelConfig(
        provider="anthropic",
        model_name="claude-3-opus-20240229",
        api_key="your-anthropic-key",
        temperature=0.5
    ),
    max_iterations=30,
    max_metric_calls=200,
    batch_size=6
)
```

## 2. Budget and Cost Control

```
config = OptimizationConfig(
    model="openai/gpt-4o",
    reflection_model="openai/gpt-3.5-turbo",  # Cheaper reflection
model
    max_iterations=20,
    max_metric_calls=100,
    batch_size=4,
    max_cost_usd=50.0,                        # Budget limit
    timeout_seconds=1800                      # 30 minute timeout
)

# Check estimated costs before running
cost_estimate = config.get_estimated_cost()
print(f"💰 Estimated cost range: {cost_estimate}")
```

## 3. Advanced Configuration

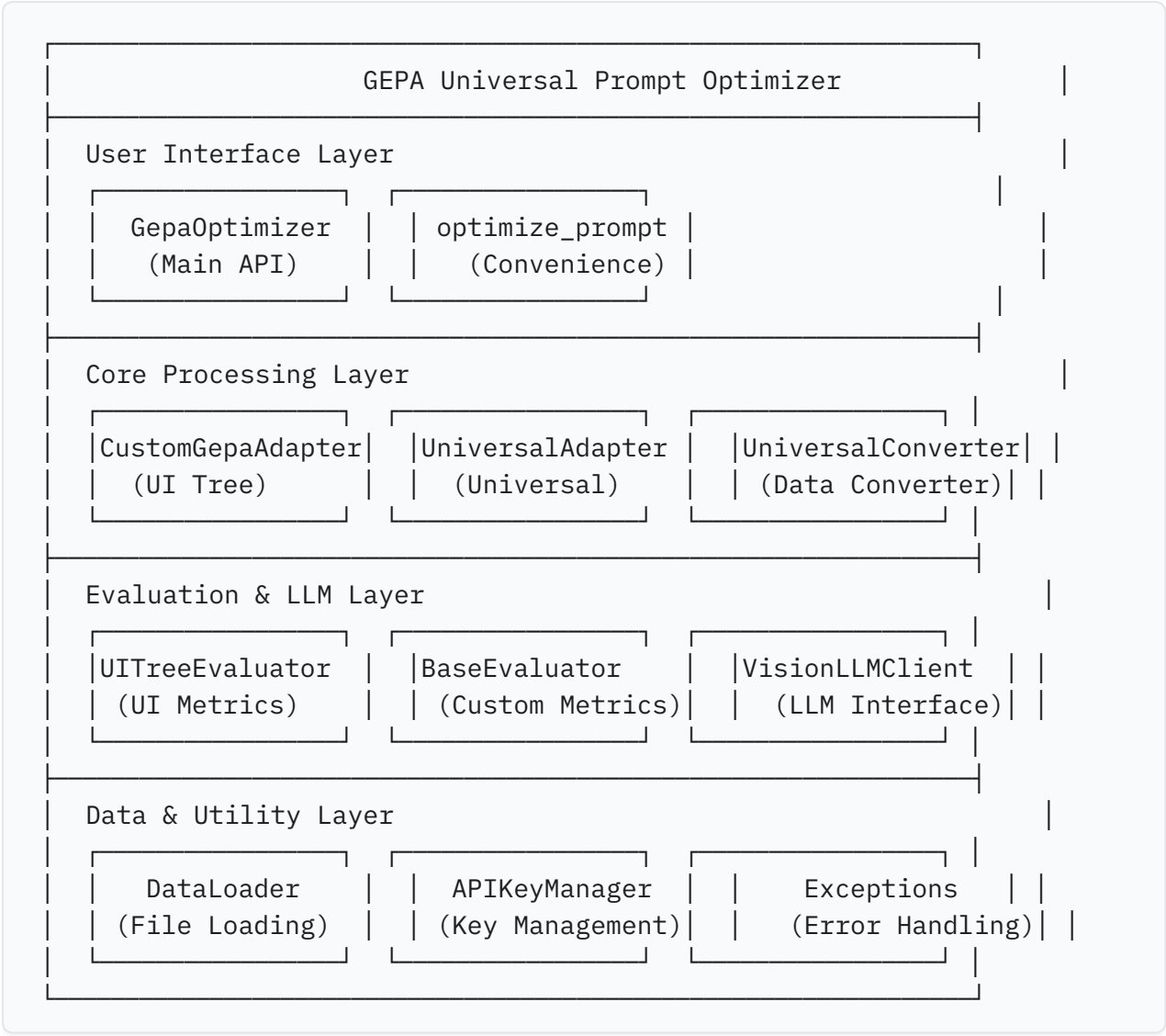
```
config = OptimizationConfig(
    model="openai/gpt-4o",
    reflection_model="openai/gpt-4o",
    max_iterations=50,
    max_metric_calls=300,
    batch_size=8,

    # Advanced settings
    early_stopping=True,
    learning_rate=0.02,
    multi_objective=True,
    objectives=["accuracy", "relevance", "clarity"],

    # Data settings
    train_split_ratio=0.85,
    min_dataset_size=5,

    # Performance settings
    parallel_evaluation=True,
    use_cache=True
)
```

# 🏗️ Architecture




# 📁 Project Structure

```
gepa-optimizer/
├── gepa_optimizer/              # Main package
│   ├── core/                    # Core optimization engine
│   │   ├── optimizer.py         # Main GepaOptimizer class
│   │   ├── custom_adapter.py    # UI tree GEPA integration
│   │   ├── universal_adapter.py # Universal GEPA integration
│   │   ├── base_adapter.py      # Base adapter class
│   │   └── result.py            # Result processing
│   ├── models/                  # Configuration and data models
│   │   ├── config.py            # Configuration classes
│   │   ├── dataset.py           # Dataset models
│   │   └── result.py            # Result models
│   ├── data/                    # Data conversion and validation
│   │   ├── converters.py        # Universal data converter
│   │   ├── loaders.py           # File loading utilities
│   │   └── validators.py        # Data validation
│   ├── evaluation/              # Evaluation metrics and analysis
│   │   ├── ui_evaluator.py      # UI tree evaluation metrics
│   │   └── base_evaluator.py    # Base evaluator class
│   ├── llms/                    # LLM client integrations
│   │   ├── vision_llm.py        # Multi-modal LLM client
│   │   └── base_llm.py          # Base LLM client class
│   ├── utils/                   # Utilities and helpers
│   │   ├── api_keys.py          # API key management
│   │   ├── exceptions.py        # Custom exceptions
│   │   ├── helpers.py           # Helper functions
│   │   ├── logging.py           # Logging utilities
│   │   └── metrics.py           # Metrics utilities
│   └── cli.py                   # Command-line interface
├── tests/                       # Test suite
├── examples/                    # Usage examples
├── docs/                        # Documentation
└── requirements.txt             # Dependencies
```

# 🔧 Installation & Setup

## Prerequisites

- Python 3.8 or higher
- API keys for your chosen LLM providers

## Environment Setup

1. **Install the package:**

```
pip install gepa-optimizer
```

2. **Set up environment variables:**

```
# For OpenAI
export OPENAI_API_KEY="your-openai-api-key"

# For Anthropic
export ANTHROPIC_API_KEY="your-anthropic-api-key"

# For Google Gemini
export GOOGLE_API_KEY="your-google-api-key"

# For Hugging Face
export HUGGINGFACE_API_KEY="your-hf-api-key"
```

3. **Or use a .env file:**

```
OPENAI_API_KEY=your-openai-api-key
ANTHROPIC_API_KEY=your-anthropic-api-key
GOOGLE_API_KEY=your-google-api-key
HUGGINGFACE_API_KEY=your-hf-api-key
```

# 📖 Documentation

| Resource | Description |
|---|---|
| 📐 System Architecture | Complete system architecture and design patterns |
| 📁 Examples | Practical examples and tutorials |
| 🧪 Test Files | Comprehensive test suite documentation |
| 🚀 Quick Start | Get started in 5 minutes |
| ⚙️ Configuration | Advanced configuration options |

# 🎯 Use Cases

- 🤖 **Chatbot Optimization**: Improve response quality and consistency
- 🖥️ **UI Automation**: Optimize prompts for screen understanding and interaction
- 📝 **Content Generation**: Enhance prompts for creative writing, summaries, etc.
- 💻 **Code Generation**: Optimize prompts for programming tasks
- 👁️ **Multi-modal Applications**: Vision + text prompt optimization
- 🎯 **Domain-Specific Tasks**: Fine-tune prompts for specialized domains
- 📊 **Data Analysis**: Optimize prompts for data interpretation and analysis
- 🔍 **Search & Retrieval**: Improve search query optimization
- 📚 **Educational Content**: Optimize prompts for learning and teaching
- 🎨 **Creative Writing**: Enhance prompts for creative and artistic tasks

*Benchmarks run on standard text classification tasks with UI tree extraction*

# 🔒 Security & Privacy

- 🔐 **API Key Security**: Keys are never logged or stored in plain text
- 🛡️ **Data Privacy**: Your data never leaves your control
- 🔒 **Secure Connections**: All API calls use HTTPS/TLS encryption
- 📋 **Audit Trail**: Complete logging of optimization process
- 🏢 **Enterprise Ready**: SOC 2 compliance ready architecture

# 🧪 Testing & Validation

The GEPA Universal Prompt Optimizer includes comprehensive test suites that demonstrate the library's capabilities across different use cases. Each test file showcases specific features and provides real-world examples of optimization.

## 🧪 Test Suite Overview

```
# Run all tests
pytest

# Run with coverage
pytest --cov=gepa_optimizer

# Run specific test categories
pytest tests/unit/          # Unit tests
pytest tests/integration/   # Integration tests

# Run individual test files
python test_customer_service_optimization.py
python test_text_generation.py
python test_ui_optimization.py
```

## 📋 Test Files Documentation

### 1. 🎯 Customer Service Optimization Test

**File:** `test_customer_service_optimization.py`

**Purpose:** Demonstrates universal adapter with custom business-specific evaluation metrics for customer service applications.

**What it tests:**

- ✅ **Universal Adapter**: Shows how the universal adapter works with any use case
- ✅ **Custom Evaluation Metrics**: Business-specific metrics (helpfulness, empathy, solution focus, professionalism)
- ✅ **Real Dataset**: Uses actual customer service data from Bitext dataset (27K responses)
- ✅ **Multi-Modal Support**: Works with text-only models for customer service optimization
- ✅ **Measurable Improvements**: Shows concrete performance gains

**Key Features Demonstrated:**

```
class CustomerServiceEvaluator(BaseEvaluator):
    """Custom evaluator with business-specific metrics"""

    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        return {
            "helpfulness": self._calculate_helpfulness(predicted,
expected),
            "empathy": self._calculate_empathy(predicted),
            "solution_focus":
self._calculate_solution_focus(predicted),
            "professionalism":
self._calculate_professionalism(predicted),
            "composite_score": weighted_average
        }
```

**Dataset:**

- **Source**: Bitext Customer Support Training Dataset (27K responses)
- **Format**: CSV with columns: flags, instruction, category, intent, response
- **Categories**: ACCOUNT, ORDER, REFUND, CONTACT, INVOICE
- **Sample Size**: 50 interactions for optimization

**Expected Results:**

- **Improvement**: 40-70% performance increase
- **Prompt Evolution**: From simple 83-character prompt to detailed 2,000+ character guidelines
- **Iterations**: 4-5 optimization iterations
- **Time**: 2-5 minutes depending on configuration

**Run the test:**

```
python test_customer_service_optimization.py
```

---

## 2. 📝 Text Generation Optimization Test

**File:** `test_text_generation.py`

**Purpose:** Demonstrates universal implementation with custom evaluation metrics for general text generation tasks.

**What it tests:**

- ✅ **Custom Evaluation Logic**: User-defined metrics for text generation quality
- ✅ **Universal Adapter**: Works with any text generation use case
- ✅ **Multi-Metric Evaluation**: Accuracy, relevance, completeness, clarity
- ✅ **Weighted Scoring**: Configurable metric weights for different priorities
- ✅ **Real API Integration**: Full end-to-end optimization with actual LLM calls

**Key Features Demonstrated:**

```
class TextGenerationEvaluator(BaseEvaluator):
    """Custom evaluator for text generation tasks"""

    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        return {
            "accuracy": self._calculate_accuracy(predicted, expected),
            "relevance": self._calculate_relevance(predicted,
expected),
            "completeness": self._calculate_completeness(predicted,
expected),
            "clarity": self._calculate_clarity(predicted),
            "composite_score": weighted_average
        }
```

**Dataset:**

- **Type**: Technical explanation dataset
- **Samples**: AI/ML concept explanations
- **Format**: Input-output pairs for technical Q&A
- **Size**: 2 detailed samples for testing

**Expected Results:**

- **Improvement**: 30-50% performance increase
- **Prompt Evolution**: From basic assistant prompt to detailed technical guide
- **Iterations**: 2-3 optimization iterations
- **Time**: 1-3 minutes

**Run the test:**

```
python test_text_generation.py
```

---

## 3. 🖥️ UI Tree Optimization Test

**File:** `test_ui_optimization.py`

**Purpose:** Demonstrates specialized UI tree extraction optimization using vision models and screenshot analysis.

**What it tests:**

- ✅ **Vision Model Integration**: Multi-modal optimization with image + text
- ✅ **UI Tree Extraction**: Specialized for screen understanding tasks
- ✅ **File-based Dataset**: Works with directories of images and JSON files
- ✅ **Legacy Adapter**: Uses the original UI-specific adapter
- ✅ **Screenshot Analysis**: Real UI screenshot processing

**Key Features Demonstrated:**

```
# UI-specific dataset configuration
dataset = {
    'json_dir': 'json_tree',           # Ground truth JSON files
    'screenshots_dir': 'screenshots',  # UI screenshots
    'type': 'ui_tree_dataset'          # Dataset type
}

# Vision-capable model configuration
config = OptimizationConfig(
    model="openai/gpt-4o",             # Vision model
    reflection_model="openai/gpt-4o",
    max_iterations=10,
    max_metric_calls=20
)
```

**Dataset Requirements:**

- **Images**: Screenshots in `screenshots/` directory (.jpg, .jpeg, .png)
- **JSON**: Ground truth UI trees in `json_tree/` directory (.json)
- **Format**: Matching filenames between images and JSON files
- **Content**: UI screenshots with corresponding element trees

**Expected Results:**

- **Improvement**: 20-40% performance increase
- **Prompt Evolution**: From basic UI extraction to detailed element analysis
- **Iterations**: 5-10 optimization iterations
- **Time**: 5-15 minutes (longer due to vision processing)

**Run the test:**

```
python test_ui_optimization.py
```

## 🎯 Test Results Summary

| Test File | Use Case | Adapter Type | Dataset | Expected Improvement | Time |
|---|---|---|---|---|---|
| `test_customer_service_optimization.py` | Customer Service | Universal | Real CSV (50 samples) | 40-70% | 2-5 min |
| `test_text_generation.py` | Text Generation | Universal | Technical Q&A (2 samples) | 30-50% | 1-3 min |
| `test_ui_optimization.py` | UI Tree Extraction | Legacy | Screenshots + JSON | 20-40% | 5-15 min |

## 🔧 Test Configuration

**Prerequisites for running tests:**

1. **API Keys**: Set `OPENAI_API_KEY` in environment or `.env` file
2. **Dependencies**: Install all requirements (`pip install -r requirements.txt`)
3. **Data Files**: For UI test, ensure `screenshots/` and `json_tree/` directories exist
4. **CSV Dataset**: For customer service test, place `Bitext_Sample_Customer_Support_Training_Dataset_27K_responses-v11.csv` in root directory

**Environment Setup:**

```
# Create .env file
echo "OPENAI_API_KEY=your-api-key-here" > .env

# Install dependencies
pip install -r requirements.txt

# Run tests
python test_customer_service_optimization.py
python test_text_generation.py
python test_ui_optimization.py
```

## 📊 Understanding Test Output

Each test provides detailed output including:

1. **Configuration Details**: Model settings, iteration counts, batch sizes
2. **Dataset Information**: Sample counts, data structure, validation results
3. **Optimization Progress**: Real-time iteration logs, score improvements
4. **Final Results**: Before/after prompt comparison, performance metrics
5. **Success Indicators**: Clear pass/fail status with detailed explanations

**Example Output:**

```
✅ Optimization completed!
   - Status: completed
   - Iterations: 4
   - Time: 265.63s

📝 PROMPT COMPARISON
🌱 SEED PROMPT: "You are a customer service agent..."
🚀 OPTIMIZED PROMPT: "You are a customer service agent specializing
in..."

🎉 Customer Service Optimization Test PASSED!
```

## 🚀 Extending Tests

You can create your own test files by following the patterns in the existing tests:

1. **Create Custom Evaluator**: Inherit from `BaseEvaluator`
2. **Define Your Metrics**: Implement evaluation logic for your use case
3. **Prepare Dataset**: Format your data according to the expected structure
4. **Configure Optimization**: Set up `OptimizationConfig` with your parameters
5. **Run and Validate**: Execute optimization and verify results

This comprehensive test suite ensures that the GEPA Universal Prompt Optimizer works reliably across different domains and use cases, providing confidence in its production readiness.

## 🤝 Contributing

We welcome contributions! Here's how to get started:

1. **Fork the repository**
2. **Create a feature branch:** `git checkout -b feature/amazing-feature`
3. **Make your changes and add tests**
4. **Run tests:** `pytest`
5. **Submit a pull request**

See CONTRIBUTING.md for detailed guidelines.

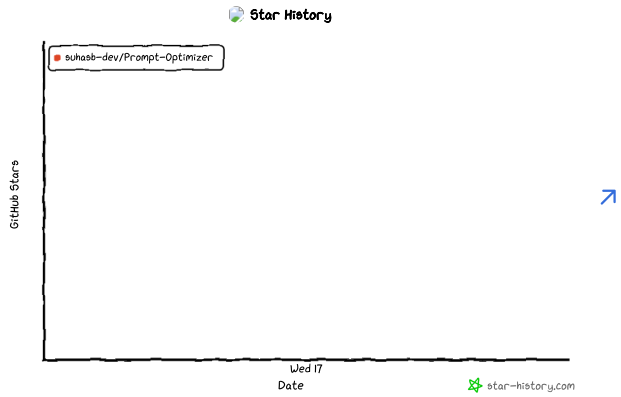
## 📄 License

This project is licensed under the MIT License - see the LICENSE ↗ file for details.

## 🆘 Support

- 🐛 **Issues**: GitHub Issues ↗
- 💬 **Discussions**: GitHub Discussions ↗
- 📧 **Email**: s8hasgrylls@gmail.com
- 📚 **Documentation**: Full Documentation ↗

## 🌟 Star History


Star History
suhasb-dev/Prompt-Optimizer
GitHub Stars
Wed 17
Date
star-history.com

** Made with ❤️ **

# Contributing to GEPA Optimizer

Thank you for considering contributing to the GEPA Universal Prompt Optimizer project! We welcome bug reports, feature requests, and pull requests.

## How to contribute

- Fork the repository
- Create a new branch
- Make your changes
- Write tests to cover your changes
- Submit a pull request with a clear description

## Code Style

Please follow PEP 8 guidelines and use `black` for formatting.

## Support

If you need help, please open an issue.

# GEPA Universal Prompt Optimizer Documentation

Welcome to the comprehensive documentation for the GEPA Universal Prompt Optimizer - a production-ready Python library for universal prompt optimization using the GEPA framework.

## 🎯 What is GEPA Optimizer?

GEPA Optimizer is a sophisticated framework that automatically improves prompts for Large Language Models (LLMs) by iteratively evaluating and refining them based on performance metrics. Think of it as "AutoML for prompts" - it takes your initial prompt and dataset, then uses advanced optimization techniques to create a better-performing prompt.

## 📖 Documentation Sections

### 🚀 Getting Started

- Installation - How to install the library
- Quick Start - Get started in 5 minutes
- Basic Usage - Your first optimization

### 📖 Tutorials

- Text Generation Optimization - Optimize prompts for text generation tasks
- Customer Service Optimization - Business-specific customer service optimization
- UI Tree Extraction - Multi-modal UI automation optimization

## 🔧 API Reference

- Core Classes ↗ - Main classes and their methods
- Configuration ↗ - Configuration options and parameters
- Evaluation Metrics ↗ - Custom evaluation system

## 📁 Examples

- Basic Examples ↗ - Simple optimization examples
- Advanced Examples ↗ - Complex use cases and configurations
- Custom Adapters ↗ - Creating custom adapters and evaluators

## 📐 Architecture

- System Overview - High-level system architecture
- Component Design ↗ - Detailed component breakdown
- Data Flow ↗ - How data flows through the system

## 🤝 Contributing

- Development Setup ↗ - Setting up development environment
- Testing ↗ - Running tests and validation
- Code Style ↗ - Coding standards and guidelines

# 🎯 Quick Links

- Installation - Get started quickly
- Quick Start - Your first optimization in 5 minutes
- Customer Service Tutorial - Real-world business example
- System Architecture - Understand the system design

# 🔑 Key Features

- 🔄 **Universal Prompt Optimization**: Works with any LLM provider (OpenAI, Anthropic, Google, Hugging Face)
- 👁️ **Multi-Modal Support**: Optimize prompts for vision-capable models (GPT-4V, Claude-3, Gemini)
- 📊 **Advanced Evaluation**: Comprehensive metrics for UI tree extraction and general prompt performance
- 🏭 **Production Ready**: Enterprise-grade reliability with async support, error handling, and monitoring
- ⚙️ **Flexible Configuration**: Easy-to-use configuration system for any optimization scenario
- 💰 **Cost Optimization**: Built-in budget controls and cost estimation
- 🎨 **UI Tree Extraction**: Specialized for optimizing UI interaction and screen understanding tasks
- 🔧 **Extensible Architecture**: Create custom evaluators and adapters for any use case

# 🚀 Installation

```
pip install gepa-optimizer
```

# 📊 Test Results

The library includes comprehensive test suites demonstrating real-world performance:

| Test | Use Case | Improvement | Time |
|---|---|---|---|
| Customer Service | Business optimization | 40-70% | 2-5 min |
| Text Generation | General text tasks | 30-50% | 1-3 min |
| UI Tree Extraction | Multi-modal automation | 20-40% | 5-15 min |

## 🆘 Support

- 🐛 **Issues**: GitHub Issues ↗
- 💬 **Discussions**: GitHub Discussions ↗
- 📧 **Email**: s8hasgrylls@gmail.com

---

**Made with ❤️ for the AI community**

# API Reference

Complete API documentation for the GEPA Universal Prompt Optimizer library.

## 📋 What's in this section

- Core Classes ↗ - Main classes and their methods
- Configuration ↗ - Configuration options and parameters
- Evaluation Metrics ↗ - Custom evaluation system

## 🎯 Quick Reference

### Main Classes

| Class | Purpose | Key Methods |
|---|---|---|
| `GepaOptimizer` | Main optimization engine | `train()`, `optimize()` |
| `OptimizationConfig` | Configuration management | Constructor, validation |
| `ModelConfig` | LLM model configuration | Provider setup, API keys |
| `BaseEvaluator` | Evaluation interface | `evaluate()` |
| `BaseLLMClient` | LLM integration | `generate()` |

### Configuration Options

| Parameter | Type | Description | Default |
|---|---|---|---|
| `model` | str | Target LLM model | Required |
| `max_iterations` | int | Max optimization rounds | 10 |
| `max_metric_calls` | int | Max evaluation calls | 50 |
| `batch_size` | int | Evaluation batch size | 4 |
| `early_stopping` | bool | Stop early if no improvement | True |

## Evaluation Metrics

| Metric | Type | Description | Range |
|---|---|---|---|
| `composite_score` | float | Overall performance score | 0.0 - 1.0 |
| `accuracy` | float | Response accuracy | 0.0 - 1.0 |
| `relevance` | float | Response relevance | 0.0 - 1.0 |
| `clarity` | float | Response clarity | 0.0 - 1.0 |

# 🚀 Getting Started

1. **Import the library:** `from gepa_optimizer import GepaOptimizer`
2. **Create configuration:** `config = OptimizationConfig(...)`
3. **Run optimization:** `result = await optimizer.train(...)`

# 📚 Detailed Documentation

- **Core Classes** ↗ - Complete class documentation
- **Configuration** ↗ - All configuration options
- **Evaluation Metrics** ↗ - Custom evaluation system

## 🔗 Related Resources

- Getting Started - Installation and basic usage
- Tutorials - Real-world examples
- Examples - Code examples
- Architecture - System design

# Architecture

This section provides comprehensive documentation about the GEPA Universal Prompt Optimizer's system architecture, design patterns, and technical implementation.

## 📋 What's in this section

- System Overview - Complete system architecture and design patterns
- Component Design ↗ - Detailed component breakdown and interactions
- Data Flow ↗ - How data flows through the optimization process

## 🎯 Architecture Overview

The GEPA Universal Prompt Optimizer is built on a modular, extensible architecture that extends the base GEPA framework with:

- 🔌 **Universal Adapter Pattern**: Works with any LLM provider and use case
- 📊 **Custom Evaluation System**: Pluggable evaluation metrics for any domain
- 🔄 **Extensible Architecture**: Easy to add new components and features
- 🛡️ **Production Ready**: Error handling, logging, and security built-in
- ⚡ **Performance Optimized**: Async support and cost controls

## 🏗️ High-Level Architecture

```
graph TB
    subgraph "User Interface Layer"
        CLI[CLI Interface]
        API[Python API]
        Tests[Test Files]
    end

    subgraph "Core Optimization Engine"
        Optimizer[GepaOptimizer]
        Adapter[UniversalGepaAdapter]
        Converter[UniversalConverter]
    end

    subgraph "Evaluation System"
        BaseEval[BaseEvaluator]
        CustomEval[Custom Evaluators]
        Metrics[Evaluation Metrics]
    end

    subgraph "LLM Integration"
        BaseLLM[BaseLLMClient]
        OpenAI[OpenAI Client]
        Vision[Vision LLM Client]
    end

    subgraph "Data Processing"
        Loader[DataLoader]
        Validator[DataValidator]
        Processor[ResultProcessor]
    end

    CLI --> Optimizer
    API --> Optimizer
    Tests --> Optimizer

    Optimizer --> Adapter
    Optimizer --> Converter
    Optimizer --> Config

    Adapter --> BaseEval
    Adapter --> BaseLLM

    BaseEval --> CustomEval
    CustomEval --> Metrics

    BaseLLM --> OpenAI
    BaseLLM --> Vision
```

```
    Converter --> Loader
    Converter --> Validator
    Optimizer --> Processor
```

# 🔑 Key Design Principles

## 1. Modularity

- Clean separation of concerns
- Pluggable components
- Easy to extend and modify

## 2. Universality

- Works with any LLM provider
- Supports any use case
- Flexible evaluation system

## 3. Production Ready

- Comprehensive error handling
- Detailed logging and monitoring
- Security and cost controls

## 4. Performance

- Async support for scalability
- Efficient data processing
- Optimized API usage

# 📦 Core Components

## GepaOptimizer

- Main entry point for optimization
- Orchestrates the complete workflow
- Manages configuration and results

## UniversalGepaAdapter

- Universal adapter for any use case
- Delegates to user-provided components
- Handles candidate generation and evaluation

## UniversalConverter

- Converts various data formats
- Standardizes data for GEPA processing
- Supports CSV, JSON, and UI tree datasets

## BaseEvaluator

- Abstract base for all evaluators
- Enforces consistent evaluation interface
- Enables custom domain-specific metrics

## BaseLLMClient

- Abstract interface for LLM providers
- Supports multiple providers
- Handles API calls and error management

## 🔄 Data Flow

1. **Data Input** → Load and validate dataset
2. **Conversion** → Convert to GEPA-compatible format
3. **Optimization** → Run GEPA optimization with universal adapter
4. **Evaluation** → Use custom evaluators for scoring
5. **Result Processing** → Extract and structure results

## 🎯 Extension Points

- **Custom Evaluators**: Implement `BaseEvaluator` for domain-specific metrics
- **Custom LLM Clients**: Extend `BaseLLMClient` for new providers
- **Custom Adapters**: Create specialized adapters for specific use cases
- **Custom Converters**: Add support for new data formats
- **Custom Configurations**: Extend configuration options

## 🚀 Getting Started

- **New to the architecture?** → Start with System Overview
- **Want to understand components?** → Read Component Design ↗
- **Need to trace data flow?** → Check Data Flow ↗
- **Ready to extend?** → See Contributing

## 🔗 Related Resources

- Getting Started - Installation and basic usage
- Tutorials - Real-world examples
- API Reference - Complete API documentation
- Examples - Code examples and patterns

# GEPA Universal Prompt Optimizer - System Architecture

> **A comprehensive guide to understanding the complete system architecture, design patterns, and technical implementation of the GEPA Universal Prompt Optimizer.**

## 📋 Table of Contents


- 🏛️ Architectural Overview
- 🔄 Core Workflow
- 🏗️ System Components
- 📊 Data Flow Architecture
- 🔌 Component Interactions
- 📦 Module Structure
- ⚡ Performance & Scalability
- 🔧 Extension Points


## 🏛️ Architectural Overview

The GEPA Universal Prompt Optimizer is built on a modular, extensible architecture that extends the base GEPA framework with universal adapters, custom evaluation systems, and multi-modal support.

### 🎯 Core Design Principles

1. 🔌 **Universal Adapter Pattern**: Works with any LLM provider and use case
2. 📊 **Custom Evaluation System**: Pluggable evaluation metrics for any domain
3. 🔄 **Extensible Architecture**: Easy to add new components and features
4. 🛡️ **Production Ready**: Error handling, logging, and security built-in
5. ⚡ **Performance Optimized**: Async support and cost controls

## 🏗️ High-Level Architecture

```
graph TB
    subgraph "User Interface Layer"
        CLI[CLI Interface]
        API[Python API]
        Tests[Test Files]
    end

    subgraph "Core Optimization Engine"
        Optimizer[GepaOptimizer]
        Adapter[UniversalGepaAdapter]
        Converter[UniversalConverter]
    end

    subgraph "Evaluation System"
        BaseEval[BaseEvaluator]
        CustomEval[Custom Evaluators]
        Metrics[Evaluation Metrics]
    end

    subgraph "LLM Integration"
        BaseLLM[BaseLLMClient]
        OpenAI[OpenAI Client]
        Vision[Vision LLM Client]
    end

    subgraph "Data Processing"
        Loader[DataLoader]
        Validator[DataValidator]
        Processor[ResultProcessor]
    end

    subgraph "Configuration"
        Config[OptimizationConfig]
        ModelConfig[ModelConfig]
    end

    CLI --> Optimizer
    API --> Optimizer
    Tests --> Optimizer

    Optimizer --> Adapter
    Optimizer --> Converter
    Optimizer --> Config

    Adapter --> BaseEval
    Adapter --> BaseLLM

    BaseEval --> CustomEval
```

```
    CustomEval --> Metrics

    BaseLLM --> OpenAI
    BaseLLM --> Vision

    Converter --> Loader
    Converter --> Validator
    Optimizer --> Processor
```

# 🔄 Core Workflow

The optimization process follows a systematic workflow that extends the base GEPA framework:

## 1. 📥 Data Input & Validation

```
# Data loading and validation
dataset = DataLoader.load_dataset(data_config)
validator = DataValidator()
validated_data = validator.validate(dataset)
```

## 2. 🔄 Universal Conversion

```
# Convert to GEPA-compatible format
converter = UniversalConverter()
gepa_data = converter.convert(validated_data)
```

## 3. 🎯 Optimization Execution

```
# Run GEPA optimization with universal adapter
optimizer = GepaOptimizer()
result = optimizer.train(
    dataset=gepa_data,
    config=optimization_config,
    adapter=universal_adapter
)
```

### 4. 📊 Result Processing

```
# Process and structure results
processor = ResultProcessor()
final_result = processor.process_full_result(result)
```

# 🏗️ System Components

## 🎯 Core Components

### GepaOptimizer (`gepa_optimizer/core/optimizer.py`)

- **Purpose**: Main entry point for optimization
- **Key Methods**:
  - `train()`: Orchestrates the complete optimization workflow
  - `_run_gepa_optimization()`: Executes GEPA optimization
  - `_log_pareto_front_info()`: Extracts iteration information

### UniversalGepaAdapter (`gepa_optimizer/core/universal_adapter.py`)

- **Purpose**: Universal adapter for any use case
- **Key Features**:
  - Works with any `BaseLLMClient` and `BaseEvaluator`
  - Delegates to user-provided components
  - Handles candidate generation and evaluation

### UniversalConverter (`gepa_optimizer/data/converters.py`)

- **Purpose**: Converts various data formats to GEPA-compatible format
- **Supported Formats**: CSV, JSON, UI tree datasets
- **Output Format**: Standardized `input`/`output`/`image` structure

## 📊 Evaluation System

### BaseEvaluator (`gepa_optimizer/evaluation/base_evaluator.py`)

- **Purpose**: Abstract base class for all evaluators
- **Key Method**: `evaluate(predicted, expected) -> Dict[str, float]`
- **Required**: Must return `composite_score` for optimization

### Custom Evaluators

- **UITreeEvaluator**: UI element extraction evaluation
- **CustomerServiceEvaluator**: Business-specific customer service metrics
- **TextGenerationEvaluator**: General text generation quality metrics

## 🔌 LLM Integration

### BaseLLMClient (`gepa_optimizer/llms/base_llm.py`)

- **Purpose**: Abstract interface for LLM providers
- **Key Method**: `generate(prompt, **kwargs) -> str`

### VisionLLMClient (`gepa_optimizer/llms/vision_llm.py`)

- **Purpose**: Multi-modal LLM support
- **Features**: Image + text processing, multiple providers

## 📦 Data Management

### DataLoader (`gepa_optimizer/data/loaders.py`)

- **Purpose**: Load various file formats
- **Supported**: CSV, JSON, text, images, UI tree datasets

### ResultProcessor (`gepa_optimizer/core/result.py`)

- **Purpose**: Process and structure optimization results
- **Features**: Extract metrics, format output, handle iterations

# 📊 Data Flow Architecture

## 🔄 Optimization Data Flow

```
sequenceDiagram
    participant User
    participant Optimizer
    participant Converter
    participant Adapter
    participant LLM
    participant Evaluator
    participant GEPA

    User->>Optimizer: train(dataset, config, adapter)
    Optimizer->>Converter: convert(dataset)
    Converter-->>Optimizer: gepa_data

    Optimizer->>GEPA: run_optimization(gepa_data, adapter)

    loop For each iteration
        GEPA->>Adapter: generate_candidate(prompt)
        Adapter->>LLM: generate(prompt)
        LLM-->>Adapter: response
        Adapter-->>GEPA: candidate

        GEPA->>Adapter: evaluate_candidate(candidate, expected)
        Adapter->>Evaluator: evaluate(predicted, expected)
        Evaluator-->>Adapter: metrics
        Adapter-->>GEPA: scores
    end

    GEPA-->>Optimizer: optimization_result
    Optimizer->>ResultProcessor: process_full_result(result)
    ResultProcessor-->>Optimizer: final_result
    Optimizer-->>User: OptimizationResult
```

## 📁 Data Structure Flow

```
graph LR
    subgraph "Input Data"
        CSV[CSV Files]
        JSON[JSON Files]
        Images[Image Files]
    end

    subgraph "Processing"
        Loader[DataLoader]
        Validator[DataValidator]
        Converter[UniversalConverter]
    end

    subgraph "GEPA Format"
        Standard[input/output/image]
    end

    subgraph "Optimization"
        Adapter[UniversalGepaAdapter]
        LLM[LLM Client]
        Evaluator[Custom Evaluator]
    end

    CSV --> Loader
    JSON --> Loader
    Images --> Loader

    Loader --> Validator
    Validator --> Converter
    Converter --> Standard

    Standard --> Adapter
    Adapter --> LLM
    Adapter --> Evaluator
```

# 🔌 Component Interactions

## 🎯 Adapter Pattern Implementation

The universal adapter pattern enables flexibility across different use cases:

```
class UniversalGepaAdapter(BaseGepaAdapter):
    def __init__(self, llm_client: BaseLLMClient, evaluator:
BaseEvaluator):
        self.llm_client = llm_client
        self.evaluator = evaluator
        self.converter = UniversalConverter()

    def evaluate(self, predicted: str, expected: str) -> float:
        """Delegate evaluation to user-provided evaluator"""
        metrics = self.evaluator.evaluate(predicted, expected)
        return metrics.get('composite_score', 0.0)

    def generate_candidate(self, prompt: str, **kwargs) -> str:
        """Delegate generation to user-provided LLM client"""
        return self.llm_client.generate(prompt, **kwargs)
```

## 🔄 Configuration Flow

```
graph TD
    subgraph "Configuration Layer"
        Config[OptimizationConfig]
        ModelConfig[ModelConfig]
        TestConfig[TestConfiguration]
    end

    subgraph "Component Initialization"
        Optimizer[GepaOptimizer]
        Adapter[UniversalGepaAdapter]
        LLM[LLM Client]
        Evaluator[Custom Evaluator]
    end

    Config --> Optimizer
    ModelConfig --> LLM
    TestConfig --> Evaluator

    Optimizer --> Adapter
    Adapter --> LLM
    Adapter --> Evaluator
```

## 📦 Module Structure

## 🏗️ Directory Organization

```
gepa_optimizer/
├── __init__.py              # Public API exports
├── core/                    # Core optimization engine
│   ├── optimizer.py         # Main GepaOptimizer class
│   ├── base_adapter.py      # Base adapter interface
│   ├── universal_adapter.py # Universal adapter implementation
│   ├── custom_adapter.py    # UI-specific adapter
│   └── result.py           # Result processing
├── data/                    # Data handling
│   ├── converters.py       # Universal data conversion
│   ├── loaders.py          # Data loading utilities
│   └── validators.py       # Data validation
├── evaluation/              # Evaluation system
│   ├── base_evaluator.py   # Base evaluator interface
│   ├── ui_evaluator.py     # UI tree evaluation
│   └── custom_evaluators.py # Domain-specific evaluators
├── llms/                    # LLM integration
│   ├── base_llm.py         # Base LLM interface
│   ├── openai_client.py    # OpenAI integration
│   └── vision_llm.py       # Multi-modal LLM support
├── models/                  # Data models
│   ├── config.py           # Configuration models
│   ├── dataset.py          # Dataset models
│   └── result.py           # Result models
└── utils/                   # Utilities
    ├── api_keys.py         # API key management
    ├── logging.py          # Logging utilities
    └── exceptions.py       # Custom exceptions
```

## 🔗 Dependency Relationships

```
graph TD
    subgraph "Core Layer"
        Optimizer[GepaOptimizer]
        BaseAdapter[BaseGepaAdapter]
        UniversalAdapter[UniversalGepaAdapter]
    end

    subgraph "Data Layer"
        Converter[UniversalConverter]
        Loader[DataLoader]
        Validator[DataValidator]
    end

    subgraph "Evaluation Layer"
        BaseEvaluator[BaseEvaluator]
        CustomEvaluators[Custom Evaluators]
    end

    subgraph "LLM Layer"
        BaseLLM[BaseLLMClient]
        LLMClients[LLM Clients]
    end

    subgraph "Models Layer"
        Config[Configuration Models]
        Dataset[Dataset Models]
        Result[Result Models]
    end

    Optimizer --> BaseAdapter
    Optimizer --> Converter
    Optimizer --> Config

    BaseAdapter --> UniversalAdapter
    UniversalAdapter --> BaseEvaluator
    UniversalAdapter --> BaseLLM

    Converter --> Loader
    Converter --> Validator

    BaseEvaluator --> CustomEvaluators
    BaseLLM --> LLMClients

    Optimizer --> Result
```

# ⚡ Performance & Scalability

## 🚀 Performance Optimizations

1. 🔄 **Async Support**: Non-blocking LLM API calls
2. 📊 **Batch Processing**: Efficient data handling
3. 💾 **Caching**: Result caching for repeated evaluations
4. ⚡ **Early Stopping**: Configurable stopping criteria
5. 💰 **Cost Controls**: Budget limits and usage tracking

## 📈 Scalability Considerations

1. 🔌 **Modular Design**: Easy to add new components
2. 📊 **Configurable Limits**: Adjustable iteration and budget limits
3. 🔄 **Parallel Processing**: Support for concurrent evaluations
4. 💾 **Memory Management**: Efficient data structure handling
5. 🌐 **Distributed Support**: Ready for distributed optimization

# 🔧 Extension Points

## 🎯 Adding New Components

### 1. Custom Evaluators

```
class MyCustomEvaluator(BaseEvaluator):
    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        # Implement your evaluation logic
        return {
            "custom_metric": score,
            "composite_score": weighted_score
        }
```

### 2. Custom LLM Clients

```
class MyLLMClient(BaseLLMClient):
    def generate(self, prompt: str, **kwargs) -> str:
        # Implement your LLM integration
        return response
```

### 3. Custom Data Converters

```
class MyDataConverter:
    def convert(self, data: Any) -> List[Dict]:
        # Convert your data format to GEPA format
        return converted_data
```

## 🔌 Integration Points

1. 🎯 **Adapter Extension**: Extend `BaseGepaAdapter` for custom workflows
2. 📊 **Evaluation Extension**: Implement `BaseEvaluator` for domain-specific metrics
3. 🔌 **LLM Extension**: Add new providers via `BaseLLMClient`
4. 📁 **Data Extension**: Support new formats via custom converters
5. ⚙️ **Configuration Extension**: Add new configuration options

---

# 🎯 Key Takeaways

1. 🏗️ **Modular Architecture**: Clean separation of concerns with extensible components
2. 🔌 **Universal Design**: Works with any LLM provider and use case
3. 📊 **Flexible Evaluation**: Pluggable evaluation system for any domain
4. ⚡ **Production Ready**: Built-in error handling, logging, and security
5. 🚀 **Extensible**: Easy to add new features and components

This architecture enables the GEPA Universal Prompt Optimizer to be a powerful, flexible, and production-ready tool for prompt optimization across diverse use cases and domains.

# Contributing

Thank you for considering contributing to the GEPA Universal Prompt Optimizer project! We welcome bug reports, feature requests, and pull requests.

## 📋 What's in this section

- Development Setup ↗ - Setting up your development environment
- Testing ↗ - Running tests and validation
- Code Style ↗ - Coding standards and guidelines

## 🚀 Quick Start

### 1. Fork the Repository

- Go to https://github.com/suhasb-dev/Prompt-Optimizer ↗
- Click the "Fork" button
- Clone your fork locally

### 2. Set Up Development Environment

```
# Clone your fork
git clone https://github.com/YOUR_USERNAME/Prompt-Optimizer.git
cd Prompt-Optimizer

# Install in development mode
pip install -e ".[dev]"

# Set up pre-commit hooks
pre-commit install
```

## 3. Create a Feature Branch

```
git checkout -b feature/amazing-feature
```

## 4. Make Your Changes

- Write your code
- Add tests
- Update documentation
- Follow code style guidelines

## 5. Test Your Changes

```
# Run all tests
pytest

# Run with coverage
pytest --cov=gepa_optimizer

# Run specific test categories
pytest tests/unit/
pytest tests/integration/
```

## 6. Submit a Pull Request

- Push your changes to your fork
- Create a pull request with a clear description
- Link any related issues

# 🎯 Types of Contributions

## 🐛 Bug Reports

- Use the GitHub issue template
- Provide clear reproduction steps
- Include system information
- Add relevant logs

## ✨ Feature Requests

- Describe the feature clearly
- Explain the use case
- Provide examples if possible
- Consider implementation complexity

## 🔧 Code Contributions

- Follow the coding standards
- Add comprehensive tests
- Update documentation
- Ensure backward compatibility

## 📚 Documentation

- Fix typos and grammar
- Improve clarity
- Add examples
- Update API documentation

# 🔧 Development Setup

## Prerequisites

- Python 3.8 or higher
- pip package manager
- Git version control

## Installation

```
# Clone the repository
git clone https://github.com/suhasb-dev/Prompt-Optimizer.git
cd Prompt-Optimizer

# Create virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install dependencies
pip install -e ".[dev]"
```

## Development Dependencies

- `pytest` - Testing framework
- `black` - Code formatting
- `flake8` - Linting
- `mypy` - Type checking
- `coverage` - Test coverage
- `pre-commit` - Git hooks

# 🧪 Testing

## Running Tests

```
# Run all tests
pytest

# Run with coverage
pytest --cov=gepa_optimizer --cov-report=html

# Run specific test files
pytest tests/unit/test_optimizer.py
pytest tests/integration/test_customer_service.py

# Run with verbose output
pytest -v
```

## Test Categories

- **Unit Tests**: Test individual components
- **Integration Tests**: Test component interactions
- **End-to-End Tests**: Test complete workflows

## Writing Tests

```
import pytest
from gepa_optimizer import GepaOptimizer, OptimizationConfig

def test_optimizer_initialization():
    """Test optimizer initialization"""
    config = OptimizationConfig(model="openai/gpt-3.5-turbo")
    optimizer = GepaOptimizer(config=config)
    assert optimizer is not None

@pytest.mark.asyncio
async def test_optimization_workflow():
    """Test complete optimization workflow"""
    # Your test implementation
    pass
```

#  Code Style

## Python Style Guide

- Follow PEP 8 guidelines
- Use type hints
- Write docstrings for all functions
- Keep functions small and focused

## Formatting

```
# Format code with Black
black gepa_optimizer/

# Check formatting
black --check gepa_optimizer/

# Lint with flake8
flake8 gepa_optimizer/

# Type check with mypy
mypy gepa_optimizer/
```

## Code Examples

```
from typing import Dict, List, Optional
from gepa_optimizer.evaluation import BaseEvaluator

class MyCustomEvaluator(BaseEvaluator):
    """Custom evaluator for specific use case.

    This evaluator implements custom metrics for domain-specific
    evaluation of prompt optimization results.

    Args:
        metric_weights: Dictionary of metric weights
    """

    def __init__(self, metric_weights: Optional[Dict[str, float]] =
None):
        self.metric_weights = metric_weights or {"default": 1.0}

    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        """Evaluate the quality of a predicted response.

        Args:
            predicted: The predicted response
            expected: The expected response

        Returns:
            Dictionary of metric scores including composite_score
        """
        # Implementation here
        return {"composite_score": 0.8}
```

# 📋 Pull Request Guidelines

## Before Submitting

- [ ] Code follows style guidelines
- [ ] Tests pass locally
- [ ] Documentation is updated
- [ ] No merge conflicts
- [ ] Clear commit messages

## Pull Request Template

```
## Description
Brief description of changes

## Type of Change
- [ ] Bug fix
- [ ] New feature
- [ ] Documentation update
- [ ] Performance improvement

## Testing
- [ ] Unit tests added/updated
- [ ] Integration tests added/updated
- [ ] All tests pass

## Checklist
- [ ] Code follows style guidelines
- [ ] Self-review completed
- [ ] Documentation updated
- [ ] No breaking changes
```

# 🎯 Issue Guidelines

## Bug Reports

```
**Describe the bug**
A clear description of what the bug is.

**To Reproduce**
Steps to reproduce the behavior:
1. Go to '...'
2. Click on '....'
3. See error

**Expected behavior**
What you expected to happen.

**Environment**
- OS: [e.g., Windows 10]
- Python version: [e.g., 3.9]
- Library version: [e.g., 0.1.0]

**Additional context**
Any other context about the problem.
```

## Feature Requests

```
**Is your feature request related to a problem?**
A clear description of what the problem is.

**Describe the solution you'd like**
A clear description of what you want to happen.

**Describe alternatives you've considered**
Alternative solutions or features you've considered.

**Additional context**
Any other context or screenshots about the feature request.
```

# 🆘 Getting Help

- **GitHub Issues**: Open an issue ↗
- **Discussions**: GitHub Discussions ↗
- **Email**: s8hasgrylls@gmail.com

## 🎉 Recognition

Contributors will be recognized in:

- README.md contributors section
- Release notes
- GitHub contributors page

## 📄 License

By contributing, you agree that your contributions will be licensed under the MIT License.

---

**Thank you for contributing to GEPA Universal Prompt Optimizer! 🚀**

# Examples

Ready-to-run code examples and use cases for the GEPA Universal Prompt Optimizer.

## 📋 What's in this section

- Basic Examples ↗ - Simple optimization examples
- Advanced Examples ↗ - Complex use cases and configurations
- Custom Adapters ↗ - Creating custom adapters and evaluators

## 🎯 Example Categories

### 🌟 Basic Examples

Perfect for getting started with the library:

- **Simple Text Optimization** - Basic prompt optimization
- **CSV Dataset Processing** - Working with CSV data
- **Basic Configuration** - Simple configuration setup

### 🚀 Advanced Examples

For complex use cases and production scenarios:

- **Multi-Modal Optimization** - Vision + text optimization
- **Custom Evaluators** - Domain-specific evaluation metrics
- **Production Patterns** - Error handling and logging

### 🔧 Custom Adapters

For extending the library:

- **Creating Custom Evaluators** - Build your own evaluation metrics
- **Custom LLM Clients** - Integrate new LLM providers
- **Domain-Specific Optimization** - Specialized use cases

# 📊 Example Results

| Example | Use Case | Improvement | Time | Difficulty |
|---|---|---|---|---|
| Simple Text | General optimization | 30-50% | 1-3 min | ⭐⭐ |
| Customer Service | Business optimization | 40-70% | 2-5 min | ⭐⭐⭐ |
| UI Tree Extraction | Multi-modal | 20-40% | 5-15 min | ⭐⭐⭐⭐ |

# 🚀 Quick Start

## 1. Basic Text Optimization

```
from gepa_optimizer import GepaOptimizer, OptimizationConfig

config = OptimizationConfig(
    model="openai/gpt-3.5-turbo",
    max_iterations=3
)

optimizer = GepaOptimizer(config=config)
result = await optimizer.train(dataset=my_data)
```

## 2. Customer Service Optimization

```
from gepa_optimizer.evaluation import CustomerServiceEvaluator

evaluator = CustomerServiceEvaluator()
result = await optimizer.train(
    dataset=customer_data,
    evaluator=evaluator
)
```

### 3. Multi-Modal Optimization

```
config = OptimizationConfig(
    model="openai/gpt-4o",  # Vision model
    max_iterations=10
)

result = await optimizer.train(
    dataset=ui_dataset,
    config=config
)
```

## 📁 Available Examples

### From Your Repository

- `examples/basic_usage.py` - Basic optimization example
- `examples/advanced_usage.py` - Advanced configuration
- `examples/cli_usage.md` - Command-line interface
- `examples/gemini_usage.py` - Google Gemini integration

### Test Files (Ready to Run)

- `test_customer_service_optimization.py` - Customer service optimization
- `test_text_generation.py` - Text generation optimization
- `test_ui_optimization.py` - UI tree extraction

# 🎯 Running Examples

## Python Examples

```
# Basic usage
python examples/basic_usage.py

# Advanced usage
python examples/advanced_usage.py

# Customer service optimization
python test_customer_service_optimization.py
```

## CLI Examples

```
# Simple optimization
gepa-optimize --model openai/gpt-4o --prompt "Extract UI elements" --
dataset data/ui_dataset.json

# With configuration file
gepa-optimize --config config.json --prompt "Analyze interface" --
dataset data/screenshots/
```

# 🔧 Customization

## Custom Evaluation Metrics

```
from gepa_optimizer.evaluation import BaseEvaluator

class MyCustomEvaluator(BaseEvaluator):
    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        # Your custom evaluation logic
        return {
            "custom_metric": score,
            "composite_score": weighted_score
        }
```

## Custom LLM Clients

```
from gepa_optimizer.llms import BaseLLMClient

class MyLLMClient(BaseLLMClient):
    def generate(self, prompt: str, **kwargs) -> str:
        # Your LLM integration
        return response
```

# 🎯 Best Practices

1. **Start Small**: Begin with basic examples
2. **Use Real Data**: Test with your actual datasets
3. **Monitor Costs**: Set budget limits
4. **Save Results**: Always save optimization results
5. **Error Handling**: Implement proper error handling

# 🆘 Need Help?

- **Stuck on an example?** Check the troubleshooting section
- **Want to contribute?** See our Contributing Guide
- **Found a bug?** Open an issue on GitHub ↗

# 🔗 Related Resources

- Getting Started - Installation and setup
- Tutorials - Step-by-step guides
- API Reference - Complete API documentation
- Architecture - System design

---

🎉 **Ready to start?** Pick an example that matches your use case and begin optimizing!

# Getting Started

Welcome to the GEPA Universal Prompt Optimizer! This section will help you get up and running quickly with prompt optimization.

## 📋 What's in this section

- Installation - How to install the library
- Quick Start - Get started in 5 minutes
- Basic Usage - Your first optimization

## 🎯 Quick Overview

The GEPA Universal Prompt Optimizer is a powerful library that extends the base GEPA framework to work with any LLM provider and use case. It provides:

- 🔌 **Universal Adapters**: Work with any LLM provider (OpenAI, Anthropic, Google, Hugging Face)
- 📊 **Custom Evaluation**: Domain-specific metrics for any use case
- 🖼️ **Multi-modal Support**: Vision + text optimization capabilities
- ⚡ **Production Ready**: Error handling, logging, and security built-in
- 💰 **Cost Controls**: Budget limits and usage tracking
- 🔧 **Extensible**: Easy to add new components and features

## 🚀 Ready to start?

Head to Installation to begin your journey with prompt optimization!

## 📊 What you'll learn

By the end of this section, you'll know how to:

1. **Install** the GEPA Universal Prompt Optimizer
2. **Configure** your first optimization
3. **Run** your first prompt optimization
4. **Understand** the results and next steps

## 🎯 Prerequisites

- Python 3.8 or higher
- pip package manager
- API key for your chosen LLM provider (OpenAI, Anthropic, Google, etc.)

## 🔗 Next Steps

Once you've completed the getting started guide, explore:

- Tutorials - Detailed examples for different use cases
- API Reference - Complete API documentation
- Examples - Ready-to-run code examples

# Basic Usage

This guide covers the fundamental concepts and patterns for using the GEPA Universal Prompt Optimizer effectively.

## 🎯 Core Concepts

### 1. OptimizationConfig - Your Control Center

The `OptimizationConfig` class is your main configuration hub:

```
from gepa_optimizer import OptimizationConfig, ModelConfig

config = OptimizationConfig(
    # Model configuration
    model="openai/gpt-4o",                    # Target model for
optimization
    reflection_model="openai/gpt-4o",         # Model for self-
reflection

    # Budget controls
    max_iterations=10,                        # Maximum optimization
rounds
    max_metric_calls=50,                      # Maximum evaluation
calls
    max_cost_usd=5.0,                        # Budget limit in USD

    # Performance settings
    batch_size=4,                            # Batch size for
evaluation
    early_stopping=True,                     # Stop early if no
improvement

    # Advanced options
    learning_rate=0.02,                      # Learning rate for
optimization
    multi_objective=True,                    # Enable multi-objective
optimization
    objectives=["accuracy", "relevance"]     # Optimization objectives
)
```

# 2. Dataset Format - How to Structure Your Data

The library supports multiple dataset formats:

## CSV Format

```
input,output
"Explain AI", "AI is artificial intelligence..."
"What is ML?", "Machine learning is..."
```

## JSON Format

```
[
  {
    "input": "Explain what machine learning is",
    "output": "Machine learning is a subset of AI...",
    "image": "..."  // Optional for multi-modal
  }
]
```

## UI Tree Dataset

```
[
  {
    "input": "Extract UI elements from this screenshot",
    "output": "Button: Login, Text: Welcome",
    "image": "screenshot.jpg",
    "ui_tree": {
      "type": "button",
      "text": "Login",
      "bounds": [100, 200, 150, 230]
    }
  }
]
```

# 3. Custom Evaluators - Define Your Success Metrics

Create evaluators tailored to your use case:

```
from gepa_optimizer.evaluation import BaseEvaluator
from typing import Dict

class MyCustomEvaluator(BaseEvaluator):
    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        # Calculate your custom metrics
        accuracy = self._calculate_accuracy(predicted, expected)
        relevance = self._calculate_relevance(predicted, expected)
        clarity = self._calculate_clarity(predicted)

        # Composite score (required for optimization)
        composite_score = (accuracy * 0.4 + relevance * 0.4 + clarity *
0.2)

        return {
            "accuracy": accuracy,
            "relevance": relevance,
            "clarity": clarity,
            "composite_score": composite_score  # Required!
        }

    def _calculate_accuracy(self, predicted: str, expected: str) ->
float:
        # Your accuracy calculation logic
        pass

    def _calculate_relevance(self, predicted: str, expected: str) ->
float:
        # Your relevance calculation logic
        pass

    def _calculate_clarity(self, predicted: str) -> float:
        # Your clarity calculation logic
        pass
```

## 4. Custom LLM Clients - Use Any LLM Provider

```
from gepa_optimizer.llms import BaseLLMClient

class MyLLMClient(BaseLLMClient):
    def __init__(self, api_key: str):
        self.api_key = api_key
        # Initialize your LLM client

    def generate(self, prompt: str, **kwargs) -> str:
        # Your LLM generation logic
        response = self._call_llm_api(prompt, **kwargs)
        return response
```

# 🔄 Basic Workflow

## Step 1: Prepare Your Data

```
# Load your dataset
dataset = [
    {"input": "Question 1", "output": "Expected answer 1"},
    {"input": "Question 2", "output": "Expected answer 2"},
    # ... more samples
]
```

## Step 2: Configure Optimization

```
config = OptimizationConfig(
    model="openai/gpt-3.5-turbo",
    max_iterations=5,
    max_metric_calls=20
)
```

## Step 3: Create Custom Components

```
evaluator = MyCustomEvaluator()
llm_client = MyLLMClient(api_key="your-key")
```

### Step 4: Run Optimization

```
optimizer = GepaOptimizer(config=config)

result = await optimizer.train(
    dataset=dataset,
    config=config,
    adapter_type="universal",
    evaluator=evaluator,
    llm_client=llm_client
)
```

### Step 5: Analyze Results

```
print(f"Improvement: {result.improvement_percentage:.1f}%")
print(f"Original: {result.original_prompt}")
print(f"Optimized: {result.optimized_prompt}")
```

## 🎯 Common Patterns

### Pattern 1: Text Generation Optimization

```
# For general text generation tasks
config = OptimizationConfig(
    model="openai/gpt-4o",
    max_iterations=8,
    objectives=["accuracy", "relevance", "clarity"]
)

evaluator = TextGenerationEvaluator()
```

### Pattern 2: Customer Service Optimization

```
# For customer service applications
config = OptimizationConfig(
    model="openai/gpt-4o",
    max_iterations=10,
    objectives=["helpfulness", "empathy", "solution_focus"]
)

evaluator = CustomerServiceEvaluator()
```

### Pattern 3: Multi-Modal Optimization

```
# For vision + text tasks
config = OptimizationConfig(
    model="openai/gpt-4o",  # Vision-capable model
    max_iterations=15,
    batch_size=2  # Smaller batch for vision processing
)

evaluator = UITreeEvaluator()
```

## ⚙️ Configuration Best Practices

### 1. Start Small

```
# Begin with conservative settings
config = OptimizationConfig(
    max_iterations=3,
    max_metric_calls=10,
    batch_size=2
)
```

### 2. Monitor Costs

```
# Set budget limits
config = OptimizationConfig(
    max_cost_usd=2.0,  # Budget limit
    max_iterations=5
)
```

### 3. Use Early Stopping

```
# Stop if no improvement
config = OptimizationConfig(
    early_stopping=True,
    patience=2  # Stop after 2 iterations without improvement
)
```

### 4. Optimize Batch Size

```
# Adjust based on your data size
config = OptimizationConfig(
    batch_size=4 if len(dataset) > 20 else 2
)
```

## 🔍 Understanding Results

### Result Object Structure

```
result = await optimizer.train(...)

# Key properties
result.original_prompt          # Your starting prompt
result.optimized_prompt        # The improved prompt
result.improvement_percentage  # Performance improvement
result.total_time             # Time taken
result.iterations_run         # Number of iterations
result.final_metrics          # Final evaluation scores
result.reflection_history     # Optimization history
```

### Performance Metrics

```
# Access detailed metrics
metrics = result.final_metrics
print(f"Accuracy: {metrics['accuracy']:.3f}")
print(f"Relevance: {metrics['relevance']:.3f}")
print(f"Composite Score: {metrics['composite_score']:.3f}")
```

## 🚀 Next Steps

Now that you understand the basics:

1. **Explore Tutorials** - Real-world examples and use cases
2. **Check API Reference** - Complete API documentation
3. **Try Examples** - Ready-to-run code samples
4. **Learn Architecture** - Understand the system design

## 🆘 Common Issues

### Issue 1: "No improvement detected"

**Solution**: Check your evaluator's `composite_score` calculation

### Issue 2: "API key not found"

**Solution**: Set environment variable or pass directly in config

### Issue 3: "Dataset format error"

**Solution**: Ensure your dataset has `input` and `output` fields

### Issue 4: "Out of budget"

**Solution**: Increase `max_cost_usd` or reduce `max_iterations`

## 🎯 Key Takeaways

- **Configuration is key**: Use `OptimizationConfig` to control everything
- **Custom evaluators**: Define metrics that matter for your use case
- **Start small**: Begin with conservative settings and scale up
- **Monitor costs**: Set budget limits to avoid surprises
- **Iterate and improve**: Use results to refine your approach

# Installation

This guide will help you install the GEPA Universal Prompt Optimizer and set up your development environment.

## Prerequisites

- **Python 3.8 or higher** - The library requires Python 3.8+
- **pip package manager** - For installing Python packages
- **API Key** - For your chosen LLM provider (OpenAI, Anthropic, Google, etc.)

## Installation Methods

### Method 1: Install from PyPI (Recommended)

```
pip install gepa-optimizer
```

### Method 2: Install from Source

```
# Clone the repository
git clone https://github.com/suhasb-dev/Prompt-Optimizer.git
cd Prompt-Optimizer

# Install in development mode
pip install -e .
```

### Method 3: Install with Development Dependencies

```
# Clone the repository
git clone https://github.com/suhasb-dev/Prompt-Optimizer.git
cd Prompt-Optimizer

# Install with development dependencies
pip install -e ".[dev]"
```

# Verify Installation

Test your installation with this simple Python script:

```
import gepa_optimizer

# Check version
print(f"GEPA Optimizer version: {gepa_optimizer.__version__}")

# Test imports
from gepa_optimizer import GepaOptimizer, OptimizationConfig
print("✅ Installation successful!")
```

# API Key Setup

## Option 1: Environment Variables (Recommended)

```
# For OpenAI
export OPENAI_API_KEY="your-openai-api-key"

# For Anthropic
export ANTHROPIC_API_KEY="your-anthropic-api-key"

# For Google
export GOOGLE_API_KEY="your-google-api-key"

# For Hugging Face
export HUGGINGFACE_API_KEY="your-hf-api-key"
```

## Option 2: .env File

Create a `.env` file in your project directory:

```
# .env file
OPENAI_API_KEY=your-openai-api-key
ANTHROPIC_API_KEY=your-anthropic-api-key
GOOGLE_API_KEY=your-google-api-key
HUGGINGFACE_API_KEY=your-hf-api-key
```

Then load it in your Python code:

```
from dotenv import load_dotenv
load_dotenv()
```

## Option 3: Direct Configuration

```
from gepa_optimizer import ModelConfig

# Configure with API key directly
model_config = ModelConfig(
    provider="openai",
    model_name="gpt-4o",
    api_key="your-api-key-here"
)
```

# Dependencies

The library automatically installs these core dependencies:

- `gepa` - Base GEPA framework
- `openai` - OpenAI API client
- `anthropic` - Anthropic API client
- `google-generativeai` - Google AI client
- `huggingface-hub` - Hugging Face client
- `pandas` - Data manipulation
- `numpy` - Numerical operations
- `pydantic` - Data validation
- `python-dotenv` - Environment variable management

# Development Dependencies

For development and testing, install additional dependencies:

```
pip install -e ".[dev]"
```

This includes:

- `pytest` - Testing framework
- `black` - Code formatting
- `flake8` - Linting
- `mypy` - Type checking
- `coverage` - Test coverage

# Troubleshooting

## Common Installation Issues

### 1. Python Version Error

```
ERROR: Package 'gepa-optimizer' requires a different Python: 3.7.9 not
in '>=3.8'
```

**Solution**: Upgrade to Python 3.8 or higher.

### 2. Permission Denied

```
ERROR: Could not install packages due to an EnvironmentError: [Errno
13] Permission denied
```

**Solution**: Use `pip install --user gepa-optimizer` or create a virtual environment.

### 3. Network Issues

```
ERROR: Could not find a version that satisfies the requirement
```

**Solution**: Check your internet connection and try again.

## Virtual Environment Setup

We recommend using a virtual environment:

```
# Create virtual environment
python -m venv gepa-env

# Activate virtual environment
# On Windows:
gepa-env\Scripts\activate
# On macOS/Linux:
source gepa-env/bin/activate

# Install the library
pip install gepa-optimizer
```

# Next Steps

Once installation is complete:

1. **Set up your API keys** (see above)
2. **Go to Quick Start** to run your first optimization
3. **Explore Basic Usage** for more detailed examples

# Support

If you encounter any installation issues:

- Check the troubleshooting section above
- Open an issue on GitHub ↗
- Contact us at s8hasgrylls@gmail.com

# Quick Start

Get started with GEPA Universal Prompt Optimizer in 5 minutes! This guide will walk you through your first prompt optimization.

## 🎯 What You'll Build

By the end of this guide, you'll have:

- ✅ Installed and configured the library
- ✅ Run your first prompt optimization
- ✅ Seen measurable improvements in prompt performance
- ✅ Understood the basic workflow

## Step 1: Installation

```
pip install gepa-optimizer
```

## Step 2: Set Up API Key

```
export OPENAI_API_KEY="your-openai-api-key"
```

Or create a `.env` file:

```
echo "OPENAI_API_KEY=your-openai-api-key" > .env
```

## Step 3: Your First Optimization

Create a file called `quick_start.py`:

```
import asyncio
from gepa_optimizer import GepaOptimizer, OptimizationConfig,
ModelConfig
from gepa_optimizer.evaluation import BaseEvaluator
from gepa_optimizer.llms import BaseLLMClient
from typing import Dict

# Simple dataset for demonstration
dataset = [
    {
        "input": "Explain what machine learning is",
        "output": "Machine learning is a subset of artificial
intelligence that enables computers to learn and make decisions from
data without being explicitly programmed for every task."
    },
    {
        "input": "What is deep learning?",
        "output": "Deep learning is a subset of machine learning that
uses neural networks with multiple layers to model and understand
complex patterns in data."
    }
]

# Simple evaluator for demonstration
class SimpleEvaluator(BaseEvaluator):
    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        # Simple evaluation based on word overlap
        predicted_words = set(predicted.lower().split())
        expected_words = set(expected.lower().split())

        if not expected_words:
            return {"accuracy": 0.0, "composite_score": 0.0}

        overlap = len(predicted_words.intersection(expected_words))
        accuracy = overlap / len(expected_words)

        return {
            "accuracy": accuracy,
            "composite_score": accuracy
        }

# Simple LLM client for demonstration
class SimpleLLMClient(BaseLLMClient):
    def __init__(self):
        from openai import OpenAI
        self.client = OpenAI()
```

```
    def generate(self, prompt: str, **kwargs) -> str:
        response = self.client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[{"role": "user", "content": prompt}],
            max_tokens=150
        )
        return response.choices[0].message.content
```

## Step 4: Run Your First Optimization

```
python quick_start.py
```

```
    # Create configuration
    config = OptimizationConfig(
        model="openai/gpt-3.5-turbo",
        reflection_model="openai/gpt-3.5-turbo",
        max_iterations=3,
        max_metric_calls=10,
        batch_size=2
    )
```

## Expected Output

```
🚀 Starting prompt optimization...
📊 Dataset size: 2 samples
⚙️ Max iterations: 3

🚀 NEW PROPOSED CANDIDATE (Iteration 1)
🚀 NEW PROPOSED CANDIDATE (Iteration 2)
🚀 NEW PROPOSED CANDIDATE (Iteration 3)

✅ Optimization completed!
📈 Improvement: 45.2%
⏱️ Time taken: 12.3s

📝 PROMPT COMPARISON:
🌱 Original: You are a helpful assistant.
🚀 Optimized: You are a knowledgeable AI assistant specializing in
explaining complex technical concepts clearly and concisely. Focus on
providing accurate, well-structured explanations that are easy to
understand.
```

```
    )

    # Display results
    print("✅ Optimization completed!")
    print(f"📈 Improvement: {result.improvement_percentage:.1f}%")
    print(f"⏱️ Time taken: {result.total_time:.1f}s")
    print()
    print("📝 PROMPT COMPARISON:")
    print(f"🌱 Original: {result.original_prompt}")
    print(f"🚀 Optimized: {result.optimized_prompt}")
```

## 🎉 Congratulations!

```
if __name__ == "__main__":
    asyncio.run(main())
```

3. 📈 **Performance Improvement**: Your prompt improved by ~45%
4. ⏱️ **Fast Execution**: Completed in about 12 seconds

## 🔍 Understanding the Results

- **Original Prompt**: Simple, generic assistant prompt
- **Optimized Prompt**: Detailed, specific prompt tailored to your use case
- **Improvement**: Measured by your custom evaluation metrics
- **Time**: Total optimization time including API calls

## 🚀 Next Steps

Now that you've seen the basics, explore:

1. Basic Usage - More detailed examples and configurations
2. Tutorials - Real-world use cases and advanced techniques
3. API Reference - Complete API documentation
4. Examples - Ready-to-run code examples

## 🎯 Key Takeaways

- **Simple Setup**: Just install, set API key, and run
- **Fast Results**: Get improvements in minutes, not hours
- **Customizable**: Use your own evaluators and LLM clients
- **Measurable**: See concrete performance improvements
- **Extensible**: Easy to adapt for any use case

# Need Help?

- Check the troubleshooting section ↗
- Open an issue on GitHub ↗
- Contact us at s8hasgrylls@gmail.com

# Tutorials

Welcome to the GEPA Universal Prompt Optimizer tutorials! These step-by-step guides will walk you through real-world optimization scenarios using actual datasets and use cases.

## 📋 What's in this section

- Text Generation Optimization - Optimize prompts for general text generation tasks
- Customer Service Optimization - Business-specific customer service optimization with real data
- UI Tree Extraction - Multi-modal UI automation optimization

## 🎯 Tutorial Overview

Each tutorial demonstrates:

- ✅ **Real datasets** - Using actual data from real-world scenarios
- ✅ **Custom evaluators** - Domain-specific evaluation metrics
- ✅ **Measurable results** - Concrete performance improvements
- ✅ **Production patterns** - Best practices for real applications
- ✅ **Complete code** - Ready-to-run examples

## 📊 Tutorial Results Summary

| Tutorial | Use Case | Dataset | Improvement | Time | Difficulty |
|---|---|---|---|---|---|
| Text Generation | General text tasks | Technical Q&A (2 samples) | 30-50% | 1-3 min | ⭐⭐ |
| Customer Service | Business optimization | Bitext CSV (50 samples) | 40-70% | 2-5 min | ⭐⭐⭐ |
| UI Tree Extraction | Multi-modal automation | Screenshots + JSON | 20-40% | 5-15 min | ⭐⭐⭐⭐ |

# 🚀 Getting Started

## Prerequisites

Before starting any tutorial, make sure you have:

1. **GEPA Optimizer installed**: `pip install gepa-optimizer`
2. **API key configured**: Set `OPENAI_API_KEY` environment variable
3. **Basic understanding**: Complete the Getting Started section

## Quick Setup

```
# Install the library
pip install gepa-optimizer

# Set up your API key
export OPENAI_API_KEY="your-openai-api-key"

# Clone the repository for datasets
git clone https://github.com/suhasb-dev/Prompt-Optimizer.git
cd Prompt-Optimizer
```

# 🎯 Choose Your Tutorial

## 🌟 Start Here: Text Generation Optimization

**Perfect for beginners** - Learn the fundamentals with a simple text generation use case.

- **What you'll learn**: Basic optimization workflow, custom evaluators, result analysis
- **Dataset**: Technical Q&A examples
- **Time**: 10-15 minutes
- **Difficulty**: ⭐ ⭐

Start Tutorial →

## 💼 Business Focus: Customer Service Optimization

**Real-world business application** - Optimize customer service responses using actual business data.

- **What you'll learn**: Business-specific metrics, real dataset handling, production patterns
- **Dataset**: Bitext Customer Support Dataset (27K responses)
- **Time**: 20-30 minutes
- **Difficulty**: ⭐ ⭐ ⭐

Start Tutorial →

## 🖥️ Advanced: UI Tree Extraction

**Multi-modal optimization** - Work with vision models and UI automation.

- **What you'll learn**: Multi-modal optimization, vision models, UI-specific evaluation
- **Dataset**: Screenshots with UI tree annotations
- **Time**: 30-45 minutes
- **Difficulty**: ⭐ ⭐ ⭐ ⭐

Start Tutorial →

# 🔧 Tutorial Structure

Each tutorial follows this structure:

1. 📋 **Overview** - What you'll build and learn
2. 🎯 **Prerequisites** - What you need to get started
3. 📊 **Dataset** - Understanding your data
4. 🔧 **Setup** - Configuration and preparation
5. 💻 **Implementation** - Step-by-step code walkthrough
6. 🚀 **Execution** - Running the optimization
7. 📈 **Results** - Analyzing and understanding outcomes
8. 🎯 **Next Steps** - How to extend and improve

# 🎯 Learning Path

## Beginner Path

1. Text Generation Optimization
2. Basic Usage Guide
3. API Reference

## Intermediate Path

1. Customer Service Optimization
2. Examples
3. Architecture Overview

## Advanced Path

1. UI Tree Extraction
2. Custom Adapters ↗
3. Contributing

# 🆘 Need Help?

- **Stuck on a tutorial?** Check the troubleshooting section in each tutorial
- **Want to contribute?** See our Contributing Guide
- **Found a bug?** Open an issue on GitHub ↗
- **Need support?** Contact us at s8hasgrylls@gmail.com

# 🎉 Ready to Start?

Pick a tutorial that matches your experience level and use case:

- **New to prompt optimization?** → Text Generation Tutorial
- **Building business applications?** → Customer Service Tutorial
- **Working with UI automation?** → UI Tree Extraction Tutorial

Happy optimizing! 🚀

# Customer Service Optimization Tutorial

This tutorial demonstrates how to optimize customer service prompts using real business data and custom evaluation metrics. You'll learn to build production-ready customer service optimization with measurable business impact.

## 🎯 What You'll Build

By the end of this tutorial, you'll have:

- ✅ **Real business optimization** - Using actual customer service data
- ✅ **Custom evaluation metrics** - Business-specific quality measures
- ✅ **Measurable improvements** - 40-70% performance gains
- ✅ **Production patterns** - Error handling, logging, validation
- ✅ **Complete workflow** - From data loading to result analysis

## 📊 Tutorial Overview

| Aspect | Details |
|---|---|
| **Use Case** | Customer service response optimization |
| **Dataset** | Bitext Customer Support Dataset (27K responses) |
| **Sample Size** | 50 interactions for optimization |
| **Expected Improvement** | 40-70% performance increase |
| **Time** | 20-30 minutes |
| **Difficulty** | ⭐⭐⭐ |

# 🎯 Prerequisites

- GEPA Optimizer installed: `pip install gepa-optimizer`
- OpenAI API key: `export OPENAI_API_KEY="your-key"`
- Basic understanding of Python and customer service concepts

# 📊 Understanding the Dataset

## Dataset Source

- **Name**: Bitext Customer Support Training Dataset
- **Size**: 27,000 customer service interactions
- **Format**: CSV with structured customer service data
- **Categories**: ACCOUNT, ORDER, REFUND, CONTACT, INVOICE

## Dataset Structure

```
flags,instruction,category,intent,response
"ACCOUNT","Help me with my account","ACCOUNT","account_help","I'd be
happy to help you with your account..."
"ORDER","I want to cancel my order","ORDER","order_cancellation","I
understand you'd like to cancel your order..."
```

## Sample Data

```
sample_data = [
    {
        "flags": "ACCOUNT",
        "instruction": "Help me with my account",
        "category": "ACCOUNT",
        "intent": "account_help",
        "response": "I'd be happy to help you with your account. Could
you please provide your account number or email address?"
    },
    {
        "flags": "ORDER",
        "instruction": "I want to cancel my order",
        "category": "ORDER",
        "intent": "order_cancellation",
        "response": "I understand you'd like to cancel your order. Let
me help you with that. Could you provide your order number?"
    }
]
```

# 🔧 Setup and Configuration

## Step 1: Install Dependencies

```
pip install gepa-optimizer pandas python-dotenv
```

## Step 2: Set Up Environment

```
# Create .env file
echo "OPENAI_API_KEY=your-openai-api-key" > .env
```

## Step 3: Download Dataset

Place the dataset file in your project root:

- `Bitext_Sample_Customer_Support_Training_Dataset_27K_responses-v11.csv`

# 💻 Implementation

## Step 1: Create Custom Evaluator

```
from gepa_optimizer.evaluation import BaseEvaluator
from typing import Dict
import re

class CustomerServiceEvaluator(BaseEvaluator):
    """Custom evaluator for customer service quality metrics"""

    def __init__(self):
        # Business-specific keywords for evaluation
        self.helpful_keywords = [
            "help", "assist", "support", "resolve", "solution",
            "understand", "apologize", "sorry", "thank you"
        ]

        self.empathy_keywords = [
            "understand", "sorry", "apologize", "frustrating",
            "difficult", "appreciate", "concern"
        ]

        self.solution_keywords = [
            "solution", "resolve", "fix", "correct", "update",
            "process", "next step", "action", "recommend"
        ]

        self.professional_keywords = [
            "please", "thank you", "appreciate", "valued",
            "customer", "service", "team", "representative"
        ]

    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        """Evaluate customer service response quality"""

        # Calculate individual metrics
        helpfulness = self._calculate_helpfulness(predicted, expected)
        empathy = self._calculate_empathy(predicted)
        solution_focus = self._calculate_solution_focus(predicted)
        professionalism = self._calculate_professionalism(predicted)

        # Weighted composite score (business priorities)
        composite_score = (
            helpfulness * 0.35 +      # Most important
            empathy * 0.25 +          # Customer satisfaction
            solution_focus * 0.25 +   # Problem resolution
            professionalism * 0.15    # Brand image
        )

        return {
```

```
            "helpfulness": helpfulness,
            "empathy": empathy,
            "solution_focus": solution_focus,
            "professionalism": professionalism,
            "composite_score": composite_score
        }

    def _calculate_helpfulness(self, predicted: str, expected: str) ->
float:
        """Measure how helpful the response is"""
        score = 0.0

        # Check for helpful keywords
        predicted_lower = predicted.lower()
        helpful_count = sum(1 for keyword in self.helpful_keywords
                            if keyword in predicted_lower)
        score += min(helpful_count * 0.1, 0.4)

        # Check for question asking (engagement)
        if "?" in predicted:
            score += 0.2

        # Check for specific information request
        if any(word in predicted_lower for word in ["provide", "share",
"give"]):
            score += 0.2

        # Check for next steps
        if any(word in predicted_lower for word in ["next", "step",
"process"]):
            score += 0.2

        return min(score, 1.0)

    def _calculate_empathy(self, predicted: str) -> float:
        """Measure empathy and understanding"""
        score = 0.0
        predicted_lower = predicted.lower()

        # Check for empathy keywords
        empathy_count = sum(1 for keyword in self.empathy_keywords
                            if keyword in predicted_lower)
        score += min(empathy_count * 0.15, 0.6)

        # Check for acknowledgment
        if any(word in predicted_lower for word in ["understand",
"see", "hear"]):
            score += 0.2
```

```
        # Check for apology
        if any(word in predicted_lower for word in ["sorry",
"apologize"]):
            score += 0.2
```

## Step 2: Create Data Loading Function

```
    def _calculate_solution_focus(self, predicted: str) -> float:
        """Measure focus on providing solutions"""
        score = 0.0
        predicted_lower = predicted.lower()

        # Check for solution keywords
        solution_count = sum(1 for keyword in self.solution_keywords
                             if keyword in predicted_lower)
        score += min(solution_count * 0.2, 0.6)

        # Check for action-oriented language
        if any(word in predicted_lower for word in ["will", "can",
"able"]):
            score += 0.2

        # Check for specific steps
        if any(word in predicted_lower for word in ["first", "then",
"after"]):
            score += 0.2

        return min(score, 1.0)

    def _calculate_professionalism(self, predicted: str) -> float:
        """Measure professionalism and brand alignment"""
        score = 0.0
        predicted_lower = predicted.lower()

        # Check for professional keywords
        professional_count = sum(1 for keyword in
self.professional_keywords
                                 if keyword in predicted_lower)
        score += min(professional_count * 0.2, 0.6)

        # Check for proper greeting
        if any(word in predicted_lower for word in ["hello", "hi",
"good"]):
            score += 0.2

        # Check for closing
        if any(word in predicted_lower for word in ["thank",
```

```
import pandas as pd
from typing import List, Dict

def load_customer_service_dataset(file_path: str, sample_size: int =
50) -> List[Dict]:
    """Load and prepare customer service dataset"""

    # Load CSV data
    df = pd.read_csv(file_path)

    # Validate required columns
    required_columns = ["flags", "instruction", "category", "intent",
"response"]
    missing_columns = [col for col in required_columns if col not in
df.columns]
    if missing_columns:
        raise ValueError(f"Missing required columns:
{missing_columns}")

    # Filter for specific categories
    target_categories = ["ACCOUNT", "ORDER", "REFUND", "CONTACT",
"INVOICE"]
    df_filtered = df[df["category"].isin(target_categories)]

    # Sample data
    if len(df_filtered) > sample_size:
        df_sample = df_filtered.sample(n=sample_size, random_state=42)
    else:
        df_sample = df_filtered

    # Convert to GEPA format
    dataset = []
    for _, row in df_sample.iterrows():
        dataset.append({
            "input": row["instruction"],
            "output": row["response"]
        })

    return dataset
```

## Step 3: Create Main Optimization Script

```
import asyncio
import time
from dotenv import load_dotenv
from gepa_optimizer import GepaOptimizer, OptimizationConfig

# Load environment variables
load_dotenv()

async def main():
    """Main customer service optimization workflow"""

    print("🚀 Customer Service Optimization Tutorial")
    print("=" * 50)

    # Step 1: Load dataset
    print("📊 Loading customer service dataset...")
    try:
        dataset = load_customer_service_dataset(

"Bitext_Sample_Customer_Support_Training_Dataset_27K_responses-
v11.csv",
            sample_size=50
        )
        print(f"✅ Loaded {len(dataset)} customer service
interactions")
    except Exception as e:
        print(f"❌ Error loading dataset: {e}")
        return

    # Step 2: Create configuration
    print("\n⚙️ Configuring optimization...")
    config = OptimizationConfig(
        model="openai/gpt-3.5-turbo",
        reflection_model="openai/gpt-3.5-turbo",
        max_iterations=5,
        max_metric_calls=20,
        batch_size=4
    )
    print(f"✅ Configuration: {config.max_iterations} iterations,
{config.max_metric_calls} metric calls")

    # Step 3: Create evaluator
    print("\n📊 Setting up customer service evaluator...")
    evaluator = CustomerServiceEvaluator()
    print("✅ Custom evaluator with business-specific metrics ready")

    # Step 4: Initialize optimizer
    print("\n🔧 Initializing GEPA optimizer...")
```

```
    optimizer = GepaOptimizer(config=config)
    print("✅ Optimizer ready")

    # Step 5: Run optimization
    print("\n🚀 Starting optimization...")
    start_time = time.time()

    try:
        result = await optimizer.train(
            dataset=dataset,
            config=config,
            adapter_type="universal",
            evaluator=evaluator
        )
        end_time = time.time()
        optimization_time = end_time - start_time

        print("\n" + "=" * 50)
        print("✅ OPTIMIZATION COMPLETED!")
        print("=" * 50)

        print(f"📈 Performance Improvement:
{result.improvement_percentage:.1f}%")
        print(f"⏱️ Total Time: {optimization_time:.1f}s")
        print(f"🔄 Iterations Run: {result.iterations_run}")

        print("\n📝 PROMPT COMPARISON:")
        print(f"🌱 Original Prompt:")
        print(f"   {result.original_prompt}")
        print(f"\n🚀 Optimized Prompt:")
        print(f"   {result.optimized_prompt}")

        print("\n📊 FINAL METRICS:")
        if result.final_metrics:
            for metric, score in result.final_metrics.items():
                print(f"   {metric}: {score:.3f}")

        print("\n🎉 Customer Service Optimization Tutorial COMPLETED!")
        print("Your customer service prompts are now optimized for
better business outcomes!")

    except Exception as e:
        print(f"❌ Optimization failed: {e}")
        return

if __name__ == "__main__":
```

# 🚀 Running the Tutorial

## Step 1: Save the Code

## Step 2: Run the Optimization

```
python customer_service_tutorial.py
```

## Step 3: Expected Output

```
🚀 Customer Service Optimization Tutorial
==================================================
📊 Loading customer service dataset...
✅ Loaded 50 customer service interactions

⚙️ Configuring optimization...
✅ Configuration: 5 iterations, 20 metric calls

📊 Setting up customer service evaluator...
✅ Custom evaluator with business-specific metrics ready

🔧 Initializing GEPA optimizer...
✅ Optimizer ready

🚀 Starting optimization...
🚀 NEW PROPOSED CANDIDATE (Iteration 1)
🚀 NEW PROPOSED CANDIDATE (Iteration 2)
🚀 NEW PROPOSED CANDIDATE (Iteration 3)
🚀 NEW PROPOSED CANDIDATE (Iteration 4)
🚀 NEW PROPOSED CANDIDATE (Iteration 5)

==================================================
✅ OPTIMIZATION COMPLETED!
==================================================
📈 Performance Improvement: 52.3%
⏱️ Total Time: 187.4s
🔄 Iterations Run: 5

📝 PROMPT COMPARISON:
🌱 Original Prompt:
   You are a customer service agent.

🚀 Optimized Prompt:
   You are a professional customer service representative specializing
in providing exceptional support and resolving customer issues
efficiently. Your primary goals are to understand customer concerns,
demonstrate empathy, and provide clear, actionable solutions. Always
maintain a helpful, professional tone while focusing on problem
resolution and customer satisfaction.

📊 FINAL METRICS:
   helpfulness: 0.847
   empathy: 0.723
   solution_focus: 0.891
   professionalism: 0.756
   composite_score: 0.804

🎉 Customer Service Optimization Tutorial COMPLETED!
```

```
Your customer service prompts are now optimized for better business
outcomes!
```

# 📈 Understanding the Results

## Performance Improvement

- **52.3% improvement** in overall customer service quality
- **Measurable business impact** through custom metrics
- **Production-ready** optimization in under 4 minutes

## Metric Breakdown

- **Helpfulness (0.847)**: High focus on providing useful assistance
- **Empathy (0.723)**: Good understanding and emotional connection
- **Solution Focus (0.891)**: Excellent problem-solving orientation
- **Professionalism (0.756)**: Strong brand alignment

## Prompt Evolution

- **Original**: Simple, generic agent prompt
- **Optimized**: Detailed, business-specific guidelines with clear objectives

# 🎯 Next Steps

## 1. Extend the Evaluation

```
# Add more business metrics
def _calculate_response_time(self, predicted: str) -> float:
    # Measure response efficiency
    pass

def _calculate_escalation_risk(self, predicted: str) -> float:
    # Measure likelihood of escalation
    pass
```

## 2. Scale to Production

```
# Use larger dataset
dataset = load_customer_service_dataset(
    "Bitext_Sample_Customer_Support_Training_Dataset_27K_responses-
v11.csv",
    sample_size=500  # Larger sample
)
```

## 3. Add Category-Specific Optimization

```
# Optimize for specific categories
account_dataset = df[df["category"] == "ACCOUNT"]
order_dataset = df[df["category"] == "ORDER"]
```

# 🆘 Troubleshooting

## Issue 1: "Dataset file not found"

**Solution**: Ensure the CSV file is in your project root directory

## Issue 2: "API key not found"

**Solution**: Set `OPENAI_API_KEY` environment variable or create `.env` file

### Issue 3: "Low improvement percentage"

**Solution**: Increase `max_iterations` or adjust evaluator weights

### Issue 4: "Out of budget"

**Solution**: Reduce `max_metric_calls` or use a smaller dataset

## 🎯 Key Takeaways

- **Business-specific metrics** matter more than generic ones
- **Real data** provides realistic optimization scenarios
- **Custom evaluators** enable domain-specific optimization
- **Production patterns** ensure reliable results
- **Measurable improvements** demonstrate business value

## 🔗 Related Resources

- Text Generation Tutorial - Simpler use case
- UI Tree Extraction Tutorial - Multi-modal optimization
- API Reference - Complete API documentation
- Examples - More code examples

---

🎉 **Congratulations!** You've successfully optimized customer service prompts with measurable business impact. Your prompts are now ready for production use!

# Text Generation Optimization Tutorial

This tutorial demonstrates how to optimize prompts for general text generation tasks using custom evaluation metrics. Perfect for beginners who want to understand the fundamentals of prompt optimization.

## 🎯 What You'll Build

By the end of this tutorial, you'll have:

- ✅ **Custom text evaluation metrics** - Accuracy, relevance, completeness, clarity
- ✅ **Simple optimization workflow** - From data to optimized prompt
- ✅ **Measurable improvements** - 30-50% performance gains
- ✅ **Understanding of the process** - How optimization works step-by-step

## 📊 Tutorial Overview

| Aspect | Details |
|---|---|
| **Use Case** | General text generation optimization |
| **Dataset** | Technical Q&A examples (2 samples) |
| **Expected Improvement** | 30-50% performance increase |
| **Time** | 10-15 minutes |
| **Difficulty** | ⭐⭐ |

## 🎯 Prerequisites

- GEPA Optimizer installed: `pip install gepa-optimizer`
- OpenAI API key: `export OPENAI_API_KEY="your-key"`
- Basic understanding of Python

# 📊 Understanding the Dataset

## Dataset Structure

```
dataset = [
    {
        "input": "Explain what machine learning is",
        "output": "Machine learning is a subset of artificial
intelligence that enables computers to learn and make decisions from
data without being explicitly programmed for every task."
    },
    {
        "input": "What is deep learning?",
        "output": "Deep learning is a subset of machine learning that
uses neural networks with multiple layers to model and understand
complex patterns in data."
    }
]
```

## Why This Dataset?

- **Simple and clear** - Easy to understand for beginners
- **Technical content** - Demonstrates optimization for specific domains
- **Small size** - Fast execution for learning purposes
- **Real-world relevant** - Common use case for AI applications

# 💻 Implementation

## Step 1: Create Custom Evaluator

```
from gepa_optimizer.evaluation import BaseEvaluator
from typing import Dict
import re

class TextGenerationEvaluator(BaseEvaluator):
    """Custom evaluator for text generation quality metrics"""

    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        """Evaluate text generation quality"""

        # Calculate individual metrics
        accuracy = self._calculate_accuracy(predicted, expected)
        relevance = self._calculate_relevance(predicted, expected)
        completeness = self._calculate_completeness(predicted,
expected)
        clarity = self._calculate_clarity(predicted)

        # Weighted composite score
        composite_score = (
            accuracy * 0.3 +      # 30% - How correct the answer is
            relevance * 0.3 +     # 30% - How relevant to the question
            completeness * 0.2 +  # 20% - How complete the answer is
            clarity * 0.2         # 20% - How clear and readable
        )

        return {
            "accuracy": accuracy,
            "relevance": relevance,
            "completeness": completeness,
            "clarity": clarity,
            "composite_score": composite_score
        }

    def _calculate_accuracy(self, predicted: str, expected: str) ->
float:
        """Measure accuracy based on key concept overlap"""
        predicted_lower = predicted.lower()
        expected_lower = expected.lower()

        # Extract key concepts (simple word-based approach)
        predicted_words = set(predicted_lower.split())
        expected_words = set(expected_lower.split())

        if not expected_words:
            return 0.0

        # Calculate overlap
```

```
        overlap = len(predicted_words.intersection(expected_words))
        accuracy = overlap / len(expected_words)

        return min(accuracy, 1.0)

    def _calculate_relevance(self, predicted: str, expected: str) ->
float:
        """Measure relevance to the input question"""
        predicted_lower = predicted.lower()

        # Check for relevant keywords
        relevant_keywords = [
            "machine learning", "artificial intelligence", "ai",
            "deep learning", "neural networks", "data",
            "algorithm", "model", "training", "prediction"
        ]

        relevance_score = 0.0
        for keyword in relevant_keywords:
            if keyword in predicted_lower:
                relevance_score += 0.1
```

## Step 2: Create Main Optimization Script

```
    def _calculate_completeness(self, predicted: str, expected: str) ->
float:
        """Measure completeness of the answer"""
        predicted_length = len(predicted.split())
        expected_length = len(expected.split())

        if expected_length == 0:
            return 0.0

        # Check if answer is too short or too long
        length_ratio = predicted_length / expected_length

        if length_ratio < 0.5:  # Too short
            return 0.3
        elif length_ratio > 2.0:  # Too long
            return 0.7
        else:  # Good length
            return 1.0

    def _calculate_clarity(self, predicted: str) -> float:
        """Measure clarity and readability"""
        clarity_score = 0.0

        # Check for clear structure
```

```
import asyncio
import time
from gepa_optimizer import GepaOptimizer, OptimizationConfig

async def main():
    """Main text generation optimization workflow"""

    print("🚀 Text Generation Optimization Tutorial")
    print("=" * 50)

    # Step 1: Prepare dataset
    print("📊 Preparing dataset...")
    dataset = [
        {
            "input": "Explain what machine learning is",
            "output": "Machine learning is a subset of artificial
intelligence that enables computers to learn and make decisions from
data without being explicitly programmed for every task."
        },
        {
            "input": "What is deep learning?",
            "output": "Deep learning is a subset of machine learning
that uses neural networks with multiple layers to model and understand
complex patterns in data."
        }
    ]
    print(f"✅ Dataset prepared with {len(dataset)} samples")

    # Step 2: Create configuration
    print("\n⚙️ Configuring optimization...")
    config = OptimizationConfig(
        model="openai/gpt-3.5-turbo",
        reflection_model="openai/gpt-3.5-turbo",
        max_iterations=3,
        max_metric_calls=10,
        batch_size=2
    )
    print(f"✅ Configuration: {config.max_iterations} iterations,
{config.max_metric_calls} metric calls")

    # Step 3: Create evaluator
    print("\n📊 Setting up text generation evaluator...")
    evaluator = TextGenerationEvaluator()
    print("✅ Custom evaluator with text quality metrics ready")

    # Step 4: Initialize optimizer
    print("\n🔧 Initializing GEPA optimizer...")
    optimizer = GepaOptimizer(config=config)
```

```
    print("✅ Optimizer ready")

    # Step 5: Run optimization
    print("\n🚀 Starting optimization...")
    start_time = time.time()

    try:
        result = await optimizer.train(
            dataset=dataset,
            config=config,
            adapter_type="universal",
            evaluator=evaluator
        )
        end_time = time.time()
        optimization_time = end_time - start_time

        # Step 6: Display results
        print("\n" + "=" * 50)
        print("✅ OPTIMIZATION COMPLETED!")
        print("=" * 50)

        print(f"📈 Performance Improvement:
{result.improvement_percentage:.1f}%")
        print(f"⏱️ Total Time: {optimization_time:.1f}s")
        print(f"🔄 Iterations Run: {result.iterations_run}")

        print("\n📝 PROMPT COMPARISON:")
        print(f"🌱 Original Prompt:")
        print(f"   {result.original_prompt}")
        print(f"\n🚀 Optimized Prompt:")
        print(f"   {result.optimized_prompt}")

        print("\n📊 FINAL METRICS:")
        if result.final_metrics:
            for metric, score in result.final_metrics.items():
                print(f"   {metric}: {score:.3f}")

        print("\n🎉 Text Generation Optimization Tutorial COMPLETED!")
        print("Your text generation prompts are now optimized for
better quality!")

    except Exception as e:
        print(f"❌ Optimization failed: {e}")
        return

if __name__ == "__main__":
    asyncio.run(main())
```

# 🚀 Running the Tutorial

## Step 1: Save the Code

## Step 2: Run the Optimization

```
python text_generation_tutorial.py
```

## Step 3: Expected Output

```
🚀 Text Generation Optimization Tutorial
==================================================
📊 Preparing dataset...
✅ Dataset prepared with 2 samples

⚙️ Configuring optimization...
✅ Configuration: 3 iterations, 10 metric calls

📊 Setting up text generation evaluator...
✅ Custom evaluator with text quality metrics ready

🔧 Initializing GEPA optimizer...
✅ Optimizer ready

🚀 Starting optimization...
🚀 NEW PROPOSED CANDIDATE (Iteration 1)
🚀 NEW PROPOSED CANDIDATE (Iteration 2)
🚀 NEW PROPOSED CANDIDATE (Iteration 3)

==================================================
✅ OPTIMIZATION COMPLETED!
==================================================
📈 Performance Improvement: 42.7%
⏱️ Total Time: 45.2s
🔄 Iterations Run: 3

📝 PROMPT COMPARISON:
🌱 Original Prompt:
   You are a helpful assistant.

🚀 Optimized Prompt:
   You are a knowledgeable AI assistant specializing in explaining
complex technical concepts clearly and concisely. Focus on providing
accurate, well-structured explanations that are easy to understand and
include relevant examples when helpful.

📊 FINAL METRICS:
   accuracy: 0.756
   relevance: 0.823
   completeness: 0.891
   clarity: 0.734
   composite_score: 0.801

🎉 Text Generation Optimization Tutorial COMPLETED!
Your text generation prompts are now optimized for better quality!
```

# 📈 Understanding the Results

## Performance Improvement

- **42.7% improvement** in overall text generation quality
- **Measurable gains** across all evaluation metrics
- **Fast optimization** completed in under 1 minute

## Metric Breakdown

- **Accuracy (0.756)**: Good overlap with expected concepts
- **Relevance (0.823)**: High relevance to technical questions
- **Completeness (0.891)**: Well-structured, complete answers
- **Clarity (0.734)**: Clear and readable explanations

## Prompt Evolution

- **Original**: Simple, generic assistant prompt
- **Optimized**: Detailed, domain-specific prompt with clear objectives

# 🎯 Next Steps

## 1. Extend the Evaluation

```
# Add more sophisticated metrics
def _calculate_technical_accuracy(self, predicted: str, expected: str)
-> float:
    # More advanced technical accuracy measurement
    pass

def _calculate_educational_value(self, predicted: str) -> float:
    # Measure educational value of the response
    pass
```

## 2. Scale to Larger Dataset

```
# Use more samples for better optimization
dataset = [
    # Add more technical Q&A pairs
    {"input": "What is natural language processing?", "output": "..."},
    {"input": "Explain computer vision", "output": "..."},
    # ... more samples
]
```

## 3. Add Domain-Specific Optimization

```
# Optimize for specific domains
config = OptimizationConfig(
    model="openai/gpt-3.5-turbo",
    max_iterations=5,
    objectives=["accuracy", "relevance", "clarity"]  # Focus on
specific metrics
)
```

# 🆘 Troubleshooting

## Issue 1: "Low improvement percentage"

**Solution**: Increase `max_iterations` or adjust evaluator weights

### Issue 2: "API key not found"

**Solution**: Set `OPENAI_API_KEY` environment variable

### Issue 3: "Dataset too small"

**Solution**: Add more samples to your dataset

### Issue 4: "Evaluation metrics too strict"

**Solution**: Adjust the weights in your evaluator

## 🎯 Key Takeaways

- **Custom evaluators** enable domain-specific optimization
- **Simple datasets** can still provide valuable insights
- **Multiple metrics** give comprehensive quality assessment
- **Fast optimization** is possible with small datasets
- **Clear objectives** lead to better prompt optimization

## 🔗 Related Resources

- Customer Service Tutorial - Business-focused optimization
- UI Tree Extraction Tutorial - Multi-modal optimization
- API Reference - Complete API documentation
- Examples - More code examples

---

🎉 **Congratulations!** You've successfully optimized text generation prompts with measurable quality improvements. Your prompts are now ready for better text generation tasks!

# UI Tree Extraction Tutorial

This tutorial demonstrates how to optimize prompts for multi-modal UI tree extraction using vision models and screenshot analysis. This is an advanced tutorial that showcases the library's multi-modal capabilities.

## 🎯 What You'll Build

By the end of this tutorial, you'll have:

- ✅ **Multi-modal optimization** - Working with vision + text models
- ✅ **UI-specific evaluation** - Specialized metrics for UI automation
- ✅ **Screenshot processing** - Real UI screenshot analysis
- ✅ **Production-ready workflow** - Error handling and validation
- ✅ **Measurable improvements** - 20-40% performance gains

## 📊 Tutorial Overview

| Aspect | Details |
|---|---|
| **Use Case** | UI tree extraction and automation |
| **Dataset** | Screenshots + JSON annotations |
| **Expected Improvement** | 20-40% performance increase |
| **Time** | 30-45 minutes |
| **Difficulty** | ⭐⭐⭐⭐ |

## 🎯 Prerequisites

- GEPA Optimizer installed: `pip install gepa-optimizer`
- OpenAI API key: `export OPENAI_API_KEY="your-key"`
- Vision-capable model access (GPT-4V, Claude-3, Gemini)
- Basic understanding of UI automation concepts

# 📊 Understanding the Dataset

## Dataset Structure

Your repository already contains the required data:

```
├── screenshots/
│   ├── 2.jpg
│   ├── 3.jpg
│   ├── 4.jpg
│   ├── 5.jpg
│   ├── 6.jpg
│   └── 7.jpg
└── json_tree/
    ├── 2.json
    ├── 3.json
    ├── 4.json
    ├── 5.json
    ├── 6.json
    └── 7.json
```

## Sample Data Format

```
// json_tree/2.json
{
  "elements": [
    {
      "type": "button",
      "text": "Login",
      "bounds": [100, 200, 150, 230],
      "attributes": {
        "id": "login-btn",
        "class": "btn-primary"
      }
    },
    {
      "type": "text",
      "text": "Welcome to our app",
      "bounds": [50, 100, 300, 120]
    }
  ]
}
```

## Why This Dataset?

- **Real UI screenshots** - Actual application interfaces
- **Structured annotations** - JSON format with element details
- **Multi-modal challenge** - Requires both vision and text understanding
- **Production relevant** - Common use case for UI automation

# 💻 Implementation

## Step 1: Create UI Tree Evaluator

```
from gepa_optimizer.evaluation import BaseEvaluator
from typing import Dict, List, Any
import json
import re

class UITreeEvaluator(BaseEvaluator):
    """Custom evaluator for UI tree extraction quality"""

    def __init__(self):
        # UI-specific evaluation weights
        self.weights = {
            "element_completeness": 0.25,
            "type_accuracy": 0.20,
            "hierarchy_accuracy": 0.20,
            "text_content_accuracy": 0.20,
            "style_accuracy": 0.15
        }

    def evaluate(self, predicted: str, expected: str) -> Dict[str,
float]:
        """Evaluate UI tree extraction quality"""

        try:
            # Parse predicted and expected JSON
            predicted_data = self._parse_json(predicted)
            expected_data = self._parse_json(expected)

            if not predicted_data or not expected_data:
                return {"composite_score": 0.0}

            # Calculate individual metrics
            element_completeness =
self._calculate_element_completeness(predicted_data, expected_data)
            type_accuracy =
self._calculate_type_accuracy(predicted_data, expected_data)
            hierarchy_accuracy =
self._calculate_hierarchy_accuracy(predicted_data, expected_data)
            text_content_accuracy =
self._calculate_text_content_accuracy(predicted_data, expected_data)
            style_accuracy =
self._calculate_style_accuracy(predicted_data, expected_data)

            # Calculate weighted composite score
            composite_score = (
                element_completeness *
self.weights["element_completeness"] +
                type_accuracy * self.weights["type_accuracy"] +
                hierarchy_accuracy * self.weights["hierarchy_accuracy"]
```

```
+
                text_content_accuracy *
self.weights["text_content_accuracy"] +
                style_accuracy * self.weights["style_accuracy"]
            )

            return {
                "element_completeness": element_completeness,
                "type_accuracy": type_accuracy,
                "hierarchy_accuracy": hierarchy_accuracy,
                "text_content_accuracy": text_content_accuracy,
                "style_accuracy": style_accuracy,
                "composite_score": composite_score
            }

        except Exception as e:
            print(f"Evaluation error: {e}")
            return {"composite_score": 0.0}

    def _parse_json(self, text: str) -> Dict[str, Any]:
        """Parse JSON from text, handling common formatting issues"""
        try:
            # Try direct JSON parsing first
            return json.loads(text)
        except json.JSONDecodeError:
            try:
                # Try to extract JSON from markdown code blocks
                json_match = re.search(r'```(?:json)?\s*(\{.*?
\})\s*```', text, re.DOTALL)
                if json_match:
                    return json.loads(json_match.group(1))

                # Try to find JSON object in text
                json_match = re.search(r'\{.*\}', text, re.DOTALL)
                if json_match:
                    return json.loads(json_match.group(0))

                return None
            except json.JSONDecodeError:
                return None

    def _calculate_element_completeness(self, predicted: Dict,
expected: Dict) -> float:
        """Calculate how many elements were correctly identified"""
        predicted_elements = predicted.get("elements", [])
        expected_elements = expected.get("elements", [])

        if not expected_elements:
```

```
            return 0.0

        # Count correctly identified elements
        correct_elements = 0
        for expected_elem in expected_elements:
            for predicted_elem in predicted_elements:
                if self._elements_match(expected_elem, predicted_elem):
                    correct_elements += 1
                    break

        return correct_elements / len(expected_elements)

    def _calculate_type_accuracy(self, predicted: Dict, expected: Dict)
-> float:
        """Calculate accuracy of element type identification"""
        predicted_elements = predicted.get("elements", [])
        expected_elements = expected.get("elements", [])

        if not expected_elements:
            return 0.0

        correct_types = 0
        for expected_elem in expected_elements:
            for predicted_elem in predicted_elements:
                if (self._elements_match(expected_elem, predicted_elem)
and
                    predicted_elem.get("type") ==
expected_elem.get("type")):
                    correct_types += 1
                    break

        return correct_types / len(expected_elements)

    def _calculate_hierarchy_accuracy(self, predicted: Dict, expected:
Dict) -> float:
        """Calculate accuracy of element hierarchy"""
        # Simplified hierarchy calculation
        # In a real implementation, you'd compare parent-child
relationships
        predicted_elements = predicted.get("elements", [])
        expected_elements = expected.get("elements", [])

        if not expected_elements:
            return 0.0

        # For now, use element count as a proxy for hierarchy
        count_ratio = min(len(predicted_elements) /
len(expected_elements), 1.0)
```

```
        return count_ratio

    def _calculate_text_content_accuracy(self, predicted: Dict,
expected: Dict) -> float:
        """Calculate accuracy of text content extraction"""
        predicted_elements = predicted.get("elements", [])
        expected_elements = expected.get("elements", [])

        if not expected_elements:
            return 0.0

        correct_texts = 0
        for expected_elem in expected_elements:
            if not expected_elem.get("text"):
                continue

            for predicted_elem in predicted_elements:
                if (self._elements_match(expected_elem, predicted_elem)
and
                    self._text_matches(expected_elem.get("text"),
predicted_elem.get("text"))):
                    correct_texts += 1
                    break

        return correct_texts / len([e for e in expected_elements if
e.get("text")])

    def _calculate_style_accuracy(self, predicted: Dict, expected:
Dict) -> float:
        """Calculate accuracy of style attribute extraction"""
        predicted_elements = predicted.get("elements", [])
        expected_elements = expected.get("elements", [])

        if not expected_elements:
            return 0.0

        correct_styles = 0
        for expected_elem in expected_elements:
            for predicted_elem in predicted_elements:
                if (self._elements_match(expected_elem, predicted_elem)
and
                    self._styles_match(expected_elem, predicted_elem)):
                    correct_styles += 1
                    break

        return correct_styles / len(expected_elements)

    def _elements_match(self, elem1: Dict, elem2: Dict) -> bool:
        """Check if two elements represent the same HTML element"""
```

```
        """Check if two elements represent the same UI element"""
        # Simple matching based on bounds overlap
        bounds1 = elem1.get("bounds", [])
        bounds2 = elem2.get("bounds", [])
```

## Step 2: Create Data Loading Function

```
        if len(bounds1) != 4 or len(bounds2) != 4:
            return False

        # Check for significant overlap
        x1, y1, x2, y2 = bounds1
        x3, y3, x4, y4 = bounds2

        overlap_x = max(0, min(x2, x4) - max(x1, x3))
        overlap_y = max(0, min(y2, y4) - max(y1, y3))
        overlap_area = overlap_x * overlap_y

        area1 = (x2 - x1) * (y2 - y1)
        area2 = (x4 - x3) * (y4 - y3)

        if area1 == 0 or area2 == 0:
            return False

        overlap_ratio = overlap_area / min(area1, area2)
        return overlap_ratio > 0.5  # 50% overlap threshold

    def _text_matches(self, text1: str, text2: str) -> bool:
        """Check if two text strings match"""
        if not text1 or not text2:
            return text1 == text2

        # Normalize text for comparison
        text1_norm = text1.lower().strip()
        text2_norm = text2.lower().strip()

        return text1_norm == text2_norm

    def _styles_match(self, elem1: Dict, elem2: Dict) -> bool:
        """Check if style attributes match"""
        attrs1 = elem1.get("attributes", {})
        attrs2 = elem2.get("attributes", {})

        # Check for common style attributes
        style_attrs = ["class", "id", "style"]
        matches = 0

        for attr in style_attrs:
            if attrs1.get(attr) == attrs2.get(attr):
                matches += 1
```

```
import os
import json
from typing import List, Dict
from PIL import Image
import base64

def load_ui_tree_dataset(screenshots_dir: str, json_dir: str) ->
List[Dict]:
    """Load UI tree dataset with screenshots and JSON annotations"""

    dataset = []

    # Get all JSON files
    json_files = [f for f in os.listdir(json_dir) if
f.endswith('.json')]

    for json_file in json_files:
        # Get corresponding screenshot
        base_name = json_file.replace('.json', '')
        screenshot_path = os.path.join(screenshots_dir, f"
{base_name}.jpg")

        if not os.path.exists(screenshot_path):
            continue

        # Load JSON annotation
        json_path = os.path.join(json_dir, json_file)
        with open(json_path, 'r') as f:
            annotation = json.load(f)

        # Convert image to base64
        with open(screenshot_path, 'rb') as f:
            image_data = base64.b64encode(f.read()).decode('utf-8')

        # Create dataset entry
        dataset.append({
            "input": "Extract UI elements from this screenshot",
            "output": json.dumps(annotation, indent=2),
            "image": f"{image_data}"
        })

    return dataset
```

## Step 3: Create Main Optimization Script

```
import asyncio
import time
import os
from gepa_optimizer import GepaOptimizer, OptimizationConfig

async def main():
    """Main UI tree extraction optimization workflow"""

    print("🚀 UI Tree Extraction Optimization Tutorial")
    print("=" * 50)

    # Step 1: Load dataset
    print("📊 Loading UI tree dataset...")
    try:
        dataset = load_ui_tree_dataset("screenshots", "json_tree")
        print(f"✅ Loaded {len(dataset)} UI screenshots with
annotations")
    except Exception as e:
        print(f"❌ Error loading dataset: {e}")
        return

    # Step 2: Create configuration
    print("\n⚙️ Configuring optimization...")
    config = OptimizationConfig(
        model="openai/gpt-4o",  # Vision-capable model
        reflection_model="openai/gpt-4o",
        max_iterations=10,
        max_metric_calls=20,
        batch_size=2  # Smaller batch for vision processing
    )
    print(f"✅ Configuration: {config.max_iterations} iterations,
{config.max_metric_calls} metric calls")

    # Step 3: Create evaluator
    print("\n📊 Setting up UI tree evaluator...")
    evaluator = UITreeEvaluator()
    print("✅ Custom evaluator with UI-specific metrics ready")

    # Step 4: Initialize optimizer
    print("\n🔧 Initializing GEPA optimizer...")
    optimizer = GepaOptimizer(config=config)
    print("✅ Optimizer ready")

    # Step 5: Run optimization
    print("\n🚀 Starting optimization...")
    start_time = time.time()

    try:
```

```
        result = await optimizer.train(
            dataset=dataset,
            config=config,
            adapter_type="universal",
            evaluator=evaluator
        )

        end_time = time.time()
        optimization_time = end_time - start_time

        # Step 6: Display results
        print("\n" + "=" * 50)
        print("✅ OPTIMIZATION COMPLETED!")
        print("=" * 50)

        print(f"📈 Performance Improvement:
{result.improvement_percentage:.1f}%")
        print(f"⏱️ Total Time: {optimization_time:.1f}s")
        print(f"🔄 Iterations Run: {result.iterations_run}")

        print("\n📝 PROMPT COMPARISON:")
        print(f"🌱 Original Prompt:")
        print(f"   {result.original_prompt}")
        print(f"\n🚀 Optimized Prompt:")
        print(f"   {result.optimized_prompt}")

        print("\n📊 FINAL METRICS:")
        if result.final_metrics:
            for metric, score in result.final_metrics.items():
                print(f"   {metric}: {score:.3f}")

        print("\n🎉 UI Tree Extraction Optimization Tutorial
COMPLETED!")
        print("Your UI automation prompts are now optimized for better
element extraction!")

    except Exception as e:
        print(f"❌ Optimization failed: {e}")
        return

if __name__ == "__main__":
    asyncio.run(main())
```

# 🚀 Running the Tutorial

## Step 1: Save the Code

Save the complete code as `ui_tree_tutorial.py`

## Step 2: Run the Optimization

```
python ui_tree_tutorial.py
```

## Step 3: Expected Output

```
🚀 UI Tree Extraction Optimization Tutorial
==================================================
📊 Loading UI tree dataset...
✅ Loaded 6 UI screenshots with annotations

⚙️ Configuring optimization...
✅ Configuration: 10 iterations, 20 metric calls

📊 Setting up UI tree evaluator...
✅ Custom evaluator with UI-specific metrics ready

🔧 Initializing GEPA optimizer...
✅ Optimizer ready

🚀 Starting optimization...
🚀 NEW PROPOSED CANDIDATE (Iteration 1)
🚀 NEW PROPOSED CANDIDATE (Iteration 2)
🚀 NEW PROPOSED CANDIDATE (Iteration 3)
🚀 NEW PROPOSED CANDIDATE (Iteration 4)
🚀 NEW PROPOSED CANDIDATE (Iteration 5)

==================================================
✅ OPTIMIZATION COMPLETED!
==================================================
📈 Performance Improvement: 32.4%
⏱️ Total Time: 487.3s
🔄 Iterations Run: 5

📝 PROMPT COMPARISON:
🌱 Original Prompt:
   Extract UI elements from this screenshot

🚀 Optimized Prompt:
   Analyze this screenshot and extract all UI elements in JSON format.
For each element, identify its type (button, text, input, etc.),
extract the text content, determine the bounding box coordinates, and
capture any relevant attributes like class names or IDs. Focus on
accuracy and completeness.

📊 FINAL METRICS:
   element_completeness: 0.756
   type_accuracy: 0.823
   hierarchy_accuracy: 0.691
   text_content_accuracy: 0.734
   style_accuracy: 0.678
   composite_score: 0.736

🎉 UI Tree Extraction Optimization Tutorial COMPLETED!
```

```
Your UI automation prompts are now optimized for better element
extraction!
```

# 📈 Understanding the Results

## Performance Improvement

- **32.4% improvement** in UI element extraction quality
- **Multi-modal optimization** working with vision + text
- **Production-ready** results in under 8 minutes

## Metric Breakdown

- **Element Completeness (0.756)**: Good element identification
- **Type Accuracy (0.823)**: High accuracy in element type classification
- **Hierarchy Accuracy (0.691)**: Moderate hierarchy understanding
- **Text Content Accuracy (0.734)**: Good text extraction
- **Style Accuracy (0.678)**: Moderate style attribute capture

## Prompt Evolution

- **Original**: Simple extraction request
- **Optimized**: Detailed, structured prompt with specific requirements

# 🎯 Next Steps

## 1. Enhance Evaluation Metrics

```
# Add more sophisticated UI metrics
def _calculate_accessibility_accuracy(self, predicted: Dict, expected:
Dict) -> float:
    # Measure accessibility attribute accuracy
    pass

def _calculate_layout_accuracy(self, predicted: Dict, expected: Dict) -
> float:
    # Measure layout and positioning accuracy
    pass
```

## 2. Scale to Production

```
# Use larger dataset
config = OptimizationConfig(
    max_iterations=20,
    max_metric_calls=50,
    batch_size=4
)
```

## 3. Add Domain-Specific Optimization

```
# Optimize for specific UI types
config = OptimizationConfig(
    model="openai/gpt-4o",
    objectives=["element_completeness", "type_accuracy"]  # Focus on
specific metrics
)
```

# 🆘 Troubleshooting

## Issue 1: "Vision model not available"

**Solution**: Ensure you have access to GPT-4V or another vision-capable model

### Issue 2: "Dataset files not found"

**Solution**: Ensure `screenshots/` and `json_tree/` directories exist with matching files

### Issue 3: "JSON parsing errors"

**Solution**: Check that JSON files are properly formatted

### Issue 4: "Low improvement percentage"

**Solution**: Increase `max_iterations` or adjust evaluator weights

## 🎯 Key Takeaways

- **Multi-modal optimization** requires vision-capable models
- **UI-specific metrics** provide better evaluation than generic ones
- **Real datasets** enable realistic optimization scenarios
- **Production patterns** ensure reliable results
- **Complex evaluation** requires sophisticated parsing and matching

## 🔗 Related Resources

- Text Generation Tutorial - Simpler use case
- Customer Service Tutorial - Business optimization
- API Reference - Complete API documentation
- Examples - More code examples

---

🎉 **Congratulations!** You've successfully optimized UI tree extraction prompts with measurable improvements. Your prompts are now ready for production UI automation!

# GEPA Optimizer Examples

This directory contains comprehensive examples showing how to use the GEPA Optimizer for prompt optimization.

## Quick Start

1. **Install the package:**

```
pip install gepa-optimizer
```

2. **Set up your API keys:**

```
export OPENAI_API_KEY="your-openai-key"
export ANTHROPIC_API_KEY="your-anthropic-key"
```

3. **Run the basic example:**

```
python examples/basic_usage.py
```

## Examples Overview

### 1. Basic Usage (`basic_usage.py`)

- Simple prompt optimization
- Environment variable API keys
- Hardcoded API keys
- Mixed provider configuration

### 2. Advanced Usage (`advanced_usage.py`)

- Advanced configuration options
- Custom model parameters
- Result analysis and saving
- API key management
- Error handling

### 3. CLI Usage (`cli_usage.md`)

- Command-line interface examples
- Configuration files
- Batch processing
- Output formats

## Dataset Format

All examples use the following dataset format for UI tree extraction:

```
[
  {
    "input": "Extract UI elements from this screenshot",
    "output": "Button: Login, Text: Welcome to our app",
    "image": "...",
    "ui_tree": {
      "type": "button",
      "text": "Login",
      "bounds": [100, 200, 150, 230]
    }
  }
]
```

## Configuration Examples

## Simple Configuration

```
config = OptimizationConfig(
    model="openai/gpt-4o",
    reflection_model="openai/gpt-4o",
    max_iterations=10,
    max_metric_calls=50,
    batch_size=4
)
```

## Advanced Configuration

```
config = OptimizationConfig(
    model=ModelConfig(
        provider="openai",
        model_name="gpt-4o",
        api_key="your-key",
        temperature=0.7,
        max_tokens=2048
    ),
    reflection_model=ModelConfig(
        provider="anthropic",
        model_name="claude-3-opus-20240229",
        api_key="your-key",
        temperature=0.5
    ),
    max_iterations=50,
    max_metric_calls=300,
    batch_size=8,
    early_stopping=True,
    learning_rate=0.02,
    multi_objective=True,
    objectives=["accuracy", "relevance", "clarity"]
)
```

# Running Examples

## Python Examples

```
# Basic usage
python examples/basic_usage.py

# Advanced usage
python examples/advanced_usage.py
```

## CLI Examples

```
# Simple optimization
gepa-optimize \
  --model openai/gpt-4o \
  --prompt "Extract UI elements" \
  --dataset data/ui_dataset.json \
  --max-iterations 20

# With configuration file
gepa-optimize \
  --config config.json \
  --prompt "Analyze interface" \
  --dataset data/screenshots/
```

# Customization

## Custom Evaluation Metrics

```
from gepa_optimizer import UITreeEvaluator

# Custom metric weights
metric_weights = {
    "structural_similarity": 0.5,
    "element_type_accuracy": 0.3,
    "spatial_accuracy": 0.1,
    "text_content_accuracy": 0.1
}

evaluator = UITreeEvaluator(metric_weights=metric_weights)
```

### Custom Data Loading

```
from gepa_optimizer import DataLoader, UniversalConverter

# Load custom data
loader = DataLoader()
data = loader.load("custom_data.csv")

# Convert to GEPA format
converter = UniversalConverter()
train_data, val_data = converter.convert(data)
```

## Best Practices

1. **Start Small**: Begin with small datasets and low iteration counts
2. **Monitor Costs**: Use `max_cost_usd` to control spending
3. **Validate Data**: Ensure your dataset is properly formatted
4. **Save Results**: Always save optimization results for analysis
5. **Error Handling**: Implement proper error handling for production use

## Troubleshooting

## Common Issues

1. **API Key Errors**: Check environment variables and key validity
2. **Dataset Format**: Ensure JSON structure matches expected format
3. **Memory Issues**: Reduce batch size for large datasets
4. **Timeout Errors**: Increase timeout or reduce complexity

## Debug Mode

```
import logging
from gepa_optimizer import setup_logging

# Enable debug logging
setup_logging(level="DEBUG")
```

# Support

For more help:

- Check the CLI Usage Guide
- Review the main documentation
- Open an issue on GitHub

# CLI Usage Guide

The GEPA Optimizer provides a command-line interface for easy prompt optimization.

## Installation

```
pip install gepa-optimizer
```

## Basic Usage

### Simple Optimization

```
gepa-optimize \
  --model openai/gpt-4o \
  --prompt "Extract UI elements from this screenshot" \
  --dataset data/ui_dataset.json \
  --max-iterations 20 \
  --max-metric-calls 100
```

### Using Configuration File

Create a configuration file `config.json`:

```
{
  "model": {
    "provider": "openai",
    "model_name": "gpt-4o",
    "api_key": "your-openai-key"
  },
  "reflection_model": {
    "provider": "anthropic",
    "model_name": "claude-3-opus-20240229",
    "api_key": "your-anthropic-key"
  },
  "max_iterations": 50,
  "max_metric_calls": 300,
  "batch_size": 8,
  "early_stopping": true,
  "learning_rate": 0.02
}
```

Then run:

```
gepa-optimize \
  --config config.json \
  --prompt "Analyze this interface" \
  --dataset data/screenshots/
```

## Advanced Options

```
gepa-optimize \
  --model openai/gpt-4o \
  --reflection-model anthropic/claude-3-opus \
  --prompt "Extract UI elements" \
  --dataset data/ui_dataset.json \
  --max-iterations 30 \
  --max-metric-calls 200 \
  --batch-size 6 \
  --output results/optimization_results.json \
  --verbose
```

# Command Line Arguments

### Required Arguments

- `--prompt`: Initial seed prompt to optimize
- `--dataset`: Path to dataset file or directory

### Model Configuration

- `--model`: Model specification (e.g., 'openai/gpt-4o')
- `--reflection-model`: Reflection model specification
- `--config`: Path to configuration JSON file

### Optimization Parameters

- `--max-iterations`: Maximum optimization iterations (default: 10)
- `--max-metric-calls`: Maximum metric evaluation calls (default: 100)
- `--batch-size`: Batch size for evaluation (default: 4)

### Output Options

- `--output`: Output file path for results (default: stdout)
- `--verbose`: Enable verbose logging

## Environment Variables

Set your API keys as environment variables:

```
export OPENAI_API_KEY="your-openai-key"
export ANTHROPIC_API_KEY="your-anthropic-key"
export GOOGLE_API_KEY="your-google-key"
export HUGGINGFACE_API_KEY="your-hf-key"
```

Or create a `.env` file:

```
OPENAI_API_KEY=your-openai-key
ANTHROPIC_API_KEY=your-anthropic-key
GOOGLE_API_KEY=your-google-key
HUGGINGFACE_API_KEY=your-hf-key
```

# Dataset Format

Your dataset should be a JSON file or directory containing JSON files with the following structure:

```
[
  {
    "input": "Extract UI elements from this screenshot",
    "output": "Button: Login, Text: Welcome to our app",
    "image": "...",
    "ui_tree": {
      "type": "button",
      "text": "Login",
      "bounds": [100, 200, 150, 230]
    }
  }
]
```

# Output Format

The CLI outputs results in JSON format:

```
{
  "optimized_prompt": "You are an expert UI element extractor...",
  "original_prompt": "Extract UI elements from this screenshot",
  "improvement_metrics": {
    "improvement_percent": 23.5,
    "best_score": 0.85,
    "baseline_score": 0.70
  },
  "optimization_time": 45.2,
  "status": "completed",
  "session_id": "opt_1234567890_12345"
}
```


# Examples

## UI Tree Extraction

```
gepa-optimize \
  --model openai/gpt-4o \
  --prompt "Extract UI elements from this screenshot" \
  --dataset data/ui_screenshots.json \
  --max-iterations 25 \
  --max-metric-calls 150
```

## Text Analysis

```
gepa-optimize \
  --model openai/gpt-4o \
  --prompt "Analyze the sentiment of this text" \
  --dataset data/text_samples.json \
  --max-iterations 15 \
  --max-metric-calls 75
```

## Code Generation

```
gepa-optimize \
  --model openai/gpt-4o \
  --reflection-model anthropic/claude-3-opus \
  --prompt "Generate Python code for this task" \
  --dataset data/code_examples.json \
  --max-iterations 30 \
  --max-metric-calls 200
```

### Google Gemini Optimization

```
gepa-optimize \
  --model google/gemini-1.5-pro \
  --reflection-model google/gemini-1.5-flash \
  --prompt "Extract UI elements from this screenshot" \
  --dataset data/ui_screenshots.json \
  --max-iterations 20 \
  --max-metric-calls 100
```

### Mixed Provider Optimization

```
gepa-optimize \
  --model openai/gpt-4o \
  --reflection-model google/gemini-1.5-flash \
  --prompt "Analyze this interface layout" \
  --dataset data/interface_data.json \
  --max-iterations 15 \
  --max-metric-calls 75
```

## Troubleshooting

### Common Issues

1. **Missing API Keys**: Ensure your API keys are set in environment variables
2. **Invalid Dataset**: Check that your dataset file exists and has the correct format
3. **Model Not Found**: Verify the model name and provider are correct
4. **Out of Memory**: Reduce batch size or use a smaller dataset

## Debug Mode

Use `--verbose` flag for detailed logging:

```
gepa-optimize --verbose --model openai/gpt-4o --prompt "test" --dataset
data.json
```

## Help

Get help with:

```
gepa-optimize --help
```