Prescient

Prescient provides a unified interface for AI providers including Ollama (local), Anthropic Claude, OpenAI GPT, and HuggingFace models. Built for prescient applications that need AI predictions with provider switching, error handling, and fallback mechanisms.

Features

- **Unified Interface**: Single API for multiple AI providers
- **Local and Cloud Support**: Ollama for local/private deployments, cloud APIs for scale
- **Embedding Generation**: Vector embeddings for semantic search and AI applications
- **Text Completion**: Chat completions with context support
- **Error Handling**: Robust error handling with automatic retries
- **Health Monitoring**: Built-in health checks for all providers
- **Flexible Configuration**: Environment variable and programmatic configuration

Supported Providers

Ollama (Local)

- **Models**: Any Ollama-compatible model (llama3.1, nomic-embed-text, etc.)
- **Capabilities**: Embeddings, Text Generation, Model Management
- **Use Case**: Privacy-focused, local deployments

Anthropic Claude

- **Models**: Claude 3 (Haiku, Sonnet, Opus)
- **Capabilities**: Text Generation only (no embeddings)
- **Use Case**: High-quality conversational AI

OpenAI

- **Models**: GPT-3.5, GPT-4, text-embedding-3-small/large
- **Capabilities**: Embeddings, Text Generation
- **Use Case**: Proven performance, wide model selection

HuggingFace

- **Models**: sentence-transformers, open-source chat models
- **Capabilities**: Embeddings, Text Generation
- **Use Case**: Open-source models, research

Installation

Add this line to your application's Gemfile:

```
gem 'prescient'
```

And then execute:

```
bundle install
```

Or install it yourself as:

```
gem install prescient
```

Configuration

Environment Variables

```
# Ollama (Local)
OLLAMA_URL=http://localhost:11434
OLLAMA_EMBEDDING_MODEL=nomic-embed-text
OLLAMA_CHAT_MODEL=llama3.1:8b

# Anthropic
ANTHROPIC_API_KEY=your_api_key
ANTHROPIC_MODEL=claude-3-haiku-20240307

# OpenAI
OPENAI_API_KEY=your_api_key
OPENAI_EMBEDDING_MODEL=text-embedding-3-small
OPENAI_CHAT_MODEL=gpt-3.5-turbo

# HuggingFace
HUGGINGFACE_API_KEY=your_api_key
HUGGINGFACE_EMBEDDING_MODEL=sentence-transformers/all-MiniLM-L6-v2
HUGGINGFACE_CHAT_MODEL=microsoft/DialoGPT-medium
```

Programmatic Configuration

```
require 'prescient'

# Configure providers
Prescient.configure do |config|
  config.default_provider = :ollama
  config.timeout = 60
  config.retry_attempts = 3
  config.retry_delay = 1.0

  # Add custom Ollama configuration
  config.add_provider(:ollama, Prescient::Ollama::Provider,
    url: 'http://localhost:11434',
    embedding_model: 'nomic-embed-text',
    chat_model: 'llama3.1:8b'
  )

  # Add Anthropic
  config.add_provider(:anthropic, Prescient::Anthropic::Provider,
```

```
    api_key: ENV['ANTHROPIC_API_KEY'],
    model: 'claude-3-haiku-20240307'
  )

  # Add OpenAI
  config.add_provider(:openai, Prescient::OpenAI::Provider,
    api_key: ENV['OPENAI_API_KEY'],
    embedding_model: 'text-embedding-3-small',
    chat_model: 'gpt-3.5-turbo'
  )
end
```

Usage

Quick Start

```
require 'prescient'

# Use default provider (Ollama)
client = Prescient.client

# Generate embeddings
embedding = client.generate_embedding("Your text here")
# => [0.1, 0.2, 0.3, ...] (768-dimensional vector)

# Generate text responses
response = client.generate_response("What is Ruby?")
puts response[:response]
# => "Ruby is a dynamic, open-source programming language..."

# Health check
health = client.health_check
puts health[:status] # => "healthy"
```

Provider-Specific Usage

```
# Use specific provider
openai_client = Prescient.client(:openai)
anthropic_client = Prescient.client(:anthropic)

# Direct method calls
embedding = Prescient.generate_embedding("text", provider: :openai)
response = Prescient.generate_response("prompt", provider: :anthropic)
```

Context-Aware Generation

```
# Generate embeddings for document chunks
documents = ["Document 1 content", "Document 2 content"]
```

```ruby
embeddings = documents.map { |doc| Prescient.generate_embedding(doc) }

# Later, find relevant context and generate response
query = "What is mentioned about Ruby?"
context_items = find_relevant_documents(query, embeddings) # Your similarity search

response = Prescient.generate_response(query, context_items,
  max_tokens: 1000,
  temperature: 0.7
)

puts response[:response]
puts "Model: " + response[:model]
puts "Provider: " + response[:provider]
```

Error Handling

```ruby
begin
  response = client.generate_response("Your prompt")
rescue Prescient::ConnectionError => e
  puts "Connection failed: #{e.message}"
rescue Prescient::RateLimitError => e
  puts "Rate limited: #{e.message}"
rescue Prescient::AuthenticationError => e
  puts "Auth failed: #{e.message}"
rescue Prescient::Error => e
  puts "General error: #{e.message}"
end
```

Health Monitoring

```ruby
# Check all providers
[:ollama, :anthropic, :openai, :huggingface].each do |provider|
  health = Prescient.health_check(provider: provider)
  puts "#{provider}: #{health[:status]}"
  puts "Ready: #{health[:ready]}" if health[:ready]
end
```

Custom Prompt Templates

Prescient allows you to customize the AI assistant's behavior through configurable prompt templates:

```ruby
Prescient.configure do |config|
  config.add_provider(:customer_service, Prescient::Provider::OpenAI,
    api_key: ENV['OPENAI_API_KEY'],
    embedding_model: 'text-embedding-3-small',
    chat_model: 'gpt-3.5-turbo',
```

```
    prompt_templates: {
      system_prompt: 'You are a friendly customer service representative.',
      no_context_template: <<~TEMPLATE.strip,
        %{ system_prompt }

        Customer Question: %{query}

        Please provide a helpful response.
      TEMPLATE
      with_context_template: <<~TEMPLATE.strip
        %{ system_prompt } Use the company info below to help answer.

        Company Information:
        %{context}

        Customer Question: %{query}

        Respond based on our company policies above.
      TEMPLATE
    }
  )
end

client = Prescient.client(:customer_service)
response = client.generate_response("What's your return policy?")
```

Template Placeholders

- `%{system_prompt}` - The system/role instruction
- `%{query}` - The user's question
- `%{context}` - Formatted context items (when provided)

Template Types

- **system_prompt** - Defines the AI's role and behavior
- **no_context_template** - Used when no context items provided
- **with_context_template** - Used when context items are provided

Examples by Use Case

Technical Documentation

```
prompt_templates: {
  system_prompt: 'You are a technical documentation assistant. Provide
detailed explanations with code examples.',
  # ... templates
}
```

Creative Writing

```
prompt_templates: {
  system_prompt: 'You are a creative writing assistant. Be imaginative and
inspiring.',
  # ... templates
}
```

See `examples/custom_prompts.rb` for complete examples.

Custom Context Configurations

Define how different data types should be formatted and which fields to use for embeddings:

```
Prescient.configure do |config|
  config.add_provider(:ecommerce, Prescient::Provider::OpenAI,
    api_key: ENV['OPENAI_API_KEY'],
    context_configs: {
      'product' => {
        fields: %w[name description price category brand],
        format: '%{ name } by %{ brand }: %{ description } - $%{ price } (%{
category })',
        embedding_fields: %w[name description category brand]
      },
      'review' => {
        fields: %w[product_name rating review_text reviewer_name],
        format: '%{ product_name } - %{ rating }/5 stars: "%{ review_text
}"',
        embedding_fields: %w[product_name review_text]
      }
    }
  )
end

# Context items with explicit type
products = [
  {
    'type' => 'product',
    'name' => 'UltraBook Pro',
    'description' => 'High-performance laptop',
    'price' => '1299.99',
    'category' => 'Laptops',
    'brand' => 'TechCorp'
  }
]

client = Prescient.client(:ecommerce)
response = client.generate_response("I need a laptop for work", products)
```

Context Configuration Options

- **fields** - Array of field names available for this context type
- **format** - Template string for displaying context items

- **embedding_fields** - Specific fields to use when generating embeddings

Automatic Context Detection

The system automatically detects context types based on YOUR configured field patterns:

1. **Explicit Type Fields**: Uses `type`, `context_type`, or `model_type` field values
2. **Field Matching**: Matches items to configured contexts based on field overlap (≥50% match required)
3. **Default Fallback**: Uses generic formatting when no context configuration matches

The system has NO hardcoded context types - it's entirely driven by your configuration!

Without Context Configuration

The system works perfectly without any context configuration - it will:

- Use intelligent fallback formatting for any hash structure
- Extract text fields for embeddings while excluding common metadata (id, timestamps, etc.)
- Provide consistent behavior across different data types

```
# No context_configs needed - works with any data!
client = Prescient.client(:default)
response = client.generate_response("Analyze this", [
  { 'title' => 'Issue', 'content' => 'Server down', 'created_at' => '2024-01-
01' },
  { 'name' => 'Alert', 'message' => 'High CPU usage', 'timestamp' => 1234567
}
])
```

See `examples/custom_contexts.rb` for complete examples.

Vector Database Integration (pgvector)

Prescient integrates seamlessly with PostgreSQL's pgvector extension for storing and searching embeddings:

Setup with Docker

The included `docker-compose.yml` provides a complete setup with PostgreSQL + pgvector:

```
# Start PostgreSQL with pgvector
docker-compose up -d postgres

# The database will automatically:
# - Install pgvector extension
# - Create tables for documents and embeddings
# - Set up optimized vector indexes
# - Insert sample data for testing
```

Database Schema

The setup creates these key tables:

- `documents` - Store original content and metadata
- `document_embeddings` - Store vector embeddings for documents
- `document_chunks` - Break large documents into searchable chunks
- `chunk_embeddings` - Store embeddings for document chunks
- `search_queries` - Track search queries and performance
- `query_results` - Store search results for analysis

Vector Search Example

```
require 'prescient'
require 'pg'

# Connect to database
db = PG.connect(
  host: 'localhost',
  port: 5432,
  dbname: 'prescient_development',
  user: 'prescient',
  password: 'prescient_password'
)

# Generate embedding for a document
client = Prescient.client(:ollama)
text = "Ruby is a dynamic programming language"
embedding = client.generate_embedding(text)

# Store embedding in database
vector_str = "[#{embedding.join(',')}]"
db.exec_params(
  "INSERT INTO document_embeddings (document_id, embedding_provider,
embedding_model, embedding_dimensions, embedding, embedding_text) VALUES ($1,
$2, $3, $4, $5, $6)",
  [doc_id, 'ollama', 'nomic-embed-text', 768, vector_str, text]
)

# Perform similarity search
query_text = "What is Ruby programming?"
query_embedding = client.generate_embedding(query_text)
query_vector = "[#{query_embedding.join(',')}]"

results = db.exec_params(
  "SELECT d.title, d.content, de.embedding <=> $1::vector AS distance
   FROM documents d
   JOIN document_embeddings de ON d.id = de.document_id
   ORDER BY de.embedding <=> $1::vector
   LIMIT 5",
  [query_vector]
)
```

Distance Functions

pgvector supports three distance functions:

- **Cosine Distance** (<=>): Best for normalized embeddings
- **L2 Distance** (<->): Euclidean distance, good general purpose
- **Inner Product** (<#>): Dot product, useful for specific cases

```
-- Cosine similarity (most common)
ORDER BY embedding <=> query_vector

-- L2 distance
ORDER BY embedding <-> query_vector

-- Inner product
ORDER BY embedding <#> query_vector
```

Vector Indexes

The setup automatically creates HNSW indexes for fast similarity search:

```
-- Example index for cosine distance
CREATE INDEX idx_embeddings_cosine
ON document_embeddings
USING hnsw (embedding vector_cosine_ops)
WITH (m = 16, ef_construction = 64);
```

Advanced Search with Filters

Combine vector similarity with metadata filtering:

```
# Search with tag filtering
results = db.exec_params(
  "SELECT d.title, de.embedding <=> $1::vector as distance
   FROM documents d
   JOIN document_embeddings de ON d.id = de.document_id
   WHERE d.metadata->'tags' ? 'programming'
   ORDER BY de.embedding <=> $1::vector
   LIMIT 5",
  [query_vector]
)

# Search with difficulty and tag filters
results = db.exec_params(
  "SELECT d.title, de.embedding <=> $1::vector as distance
   FROM documents d
   JOIN document_embeddings de ON d.id = de.document_id
   WHERE d.metadata->>'difficulty' = 'beginner'
     AND d.metadata->'tags' ?| $2::text[]
   ORDER BY de.embedding <=> $1::vector
   LIMIT 5",
  [query_vector, ['ruby', 'programming']]
)
```

Performance Optimization

Index Configuration

For large datasets, tune HNSW parameters:

```
-- High accuracy (slower build, more memory)
WITH (m = 32, ef_construction = 128)

-- Fast build (lower accuracy, less memory)
WITH (m = 8, ef_construction = 32)

-- Balanced (recommended default)
WITH (m = 16, ef_construction = 64)
```

Query Performance

```
-- Set ef_search for query-time accuracy/speed tradeoff
SET hnsw.ef_search = 100;  -- Higher = more accurate, slower

-- Use EXPLAIN ANALYZE to optimize queries
EXPLAIN ANALYZE
SELECT * FROM document_embeddings
ORDER BY embedding <=> '[0.1,0.2,...]'::vector
LIMIT 10;
```

Chunking Strategy

For large documents, use chunking for better search granularity:

```
def chunk_document(text, chunk_size: 500, overlap: 50)
  chunks = []
  start = 0

  while start < text.length
    end_pos = [start + chunk_size, text.length].min
    chunk = text[start...end_pos]
    chunks << chunk
    start += chunk_size - overlap
  end

  chunks
end

# Generate embeddings for each chunk
chunks = chunk_document(document.content)
chunks.each_with_index do |chunk, index|
  embedding = client.generate_embedding(chunk)
```

```
  # Store chunk and embedding...
end
```

Example Usage

Run the complete vector search example:

```
# Start services
docker-compose up -d postgres ollama

# Run example
DB_HOST=localhost ruby examples/vector_search.rb
```

The example demonstrates:

- Document embedding generation and storage
- Similarity search with different distance functions
- Metadata filtering and advanced queries
- Performance comparison between approaches

Advanced Usage

Custom Provider Implementation

```
class MyCustomProvider < Prescient::BaseProvider
  def generate_embedding(text, **options)
    # Your implementation
  end

  def generate_response(prompt, context_items = [], **options)
    # Your implementation
  end

  def health_check
    # Your implementation
  end

  protected

  def validate_configuration!
    # Validate required options
  end
end

# Register your provider
Prescient.configure do |config|
  config.add_provider(:mycustom, MyCustomProvider,
    api_key: 'your_key',
    model: 'your_model'
  )
end
```

Provider Information

```
client = Prescient.client(:ollama)
info = client.provider_info

puts info[:name]      # => :ollama
puts info[:class]     # => "Prescient::Ollama::Provider"
puts info[:available] # => true
puts info[:options]   # => { ... } (excluding sensitive data)
```

Provider-Specific Features

Ollama

- Model management: `pull_model`, `list_models`
- Local deployment support
- No API costs

Anthropic

- High-quality responses
- No embedding support (use with OpenAI/HuggingFace for embeddings)

OpenAI

- Multiple embedding model sizes
- Latest GPT models
- Reliable performance

HuggingFace

- Open-source models
- Research-friendly
- Free tier available

Docker Setup (Recommended for Ollama)

The easiest way to get started with Prescient and Ollama is using Docker Compose:

Hardware Requirements

Before starting, ensure your system meets the minimum requirements for running Ollama:

Minimum Requirements:

- **CPU**: 4+ cores (x86_64 or ARM64)
- **RAM**: 8GB+ (16GB recommended)
- **Storage**: 10GB+ free space for models
- **OS**: Linux, macOS, or Windows with Docker

Model-Specific Requirements:

Model	RAM Required	Storage	Notes

Model	RAM Required	Storage	Notes
`nomic-embed-text`	1GB	274MB	Embedding model
`llama3.1:8b`	8GB	4.7GB	Chat model (8B parameters)
`llama3.1:70b`	64GB+	40GB	Large chat model (70B parameters)
`codellama:7b`	8GB	3.8GB	Code generation model

Performance Recommendations:

- **SSD Storage**: Significantly faster model loading
- **GPU (Optional)**: NVIDIA GPU with 8GB+ VRAM for acceleration
- **Network**: Stable internet for initial model downloads
- **Docker**: 4GB+ memory limit configured

GPU Acceleration (Optional):

- **NVIDIA GPU**: RTX 3060+ with 8GB+ VRAM recommended
- **CUDA**: Version 11.8+ required
- **Docker**: NVIDIA Container Toolkit installed
- **Performance**: 3-10x faster inference with compatible models

> **Tip**: Start with smaller models like `llama3.1:8b` and upgrade based on your hardware capabilities and performance needs.

Quick Start with Docker

1. **Start Ollama service:**

```
docker-compose up -d ollama
```

2. **Pull required models:**

```
# Automatic setup
docker-compose up ollama-init

# Or manual setup
./scripts/setup-ollama-models.sh
```

3. **Run examples:**

```
# Set environment variable
export OLLAMA_URL=http://localhost:11434

# Run examples
ruby examples/custom_contexts.rb
```

Docker Compose Services

The included `docker-compose.yml` provides:

- **ollama**: Ollama AI service with persistent model storage
- **ollama-init**: Automatically pulls required models on startup
- **redis**: Optional caching layer for embeddings
- **prescient-app**: Example Ruby application container

Configuration Options

```
# docker-compose.yml environment variables
services:
  ollama:
    ports:
      - "11434:11434" # Ollama API port
    volumes:
      - ollama_data:/root/.ollama # Persist models
    environment:
      - OLLAMA_HOST=0.0.0.0
      - OLLAMA_ORIGINS=*
```

GPU Support (Optional)

For GPU acceleration, uncomment the GPU configuration in `docker-compose.yml`:

```
services:
  ollama:
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]
```

Environment Variables

```
# Ollama Configuration
OLLAMA_URL=http://localhost:11434
OLLAMA_EMBEDDING_MODEL=nomic-embed-text
OLLAMA_CHAT_MODEL=llama3.1:8b

# Optional: Other AI providers
OPENAI_API_KEY=your_key_here
ANTHROPIC_API_KEY=your_key_here
HUGGINGFACE_API_KEY=your_key_here
```

Model Management

```
# Check available models
curl http://localhost:11434/api/tags

# Pull a specific model
curl -X POST http://localhost:11434/api/pull \
  -H "Content-Type: application/json" \
  -d '{ "name": "llama3.1:8b"}'

# Health check
curl http://localhost:11434/api/version
```

Production Deployment

For production use:

1. Use specific image tags instead of `latest`
2. Configure proper resource limits
3. Set up monitoring and logging
4. Use secrets management for API keys
5. Configure backups for model data

Troubleshooting

Common Issues:

Out of Memory Errors:

```
# Check available memory
free -h

# Increase Docker memory limit (Docker Desktop)
# Settings > Resources > Memory: 8GB+

# Use smaller models if hardware limited
OLLAMA_CHAT_MODEL=llama3.1:7b ruby examples/custom_contexts.rb
```

Slow Model Loading:

```
# Check disk I/O
iostat -x 1

# Move Docker data to SSD if on HDD
# Docker Desktop: Settings > Resources > Disk image location
```

Model Download Failures:

```
# Check disk space
df -h
```

```
# Manually pull models with retry
docker exec prescient-ollama ollama pull llama3.1:8b
```

GPU Not Detected:

```
# Check NVIDIA Docker runtime
docker run --rm --gpus all nvidia/cuda:11.8-base nvidia-smi

# Install NVIDIA Container Toolkit if missing
# https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html
```

Performance Monitoring:

```
# Monitor resource usage
docker stats prescient-ollama

# Check Ollama logs
docker logs prescient-ollama

# Test API response time
time curl -X POST http://localhost:11434/api/generate \
  -H "Content-Type: application/json" \
  -d '{ "model": "llama3.1:8b", "prompt": "Hello", "stream": false}'
```

Testing

The gem includes comprehensive test coverage:

```
bundle exec rspec
```

Development

After checking out the repo, run:

```
bundle install
```

To install this gem onto your local machine:

```
bundle exec rake install
```

Contributing

1. Fork it
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Commit your changes (`git commit -am 'Add some feature'`)
4. Push to the branch (`git push origin my-new-feature`)
5. Create a new Pull Request

License

The gem is available as open source under the terms of the MIT License.

Roadmap

Version 0.2.0 (Planned)

- **MariaDB Vector Support**: Integration with MariaDB using external vector databases
- **Hybrid Database Architecture**: Support for MariaDB + Milvus/Qdrant combinations
- **Vector Database Adapters**: Pluggable adapters for different vector storage backends
- **Enhanced Chunking Strategies**: Smart document splitting with multiple algorithms
- **Search Result Ranking**: Advanced scoring and re-ranking capabilities

Version 0.3.0 (Future)

- **Streaming Responses**: Real-time response streaming for chat applications
- **Multi-Model Ensembles**: Combine responses from multiple AI providers
- **Advanced Analytics**: Search performance insights and usage analytics
- **Cloud Provider Integration**: Direct support for Pinecone, Weaviate, etc.

Changelog

Version 0.1.0

- Initial release
- Support for Ollama, Anthropic, OpenAI, and HuggingFace
- Unified interface for embeddings and text generation
- Comprehensive error handling and retry logic
- Health monitoring capabilities
- PostgreSQL pgvector integration with complete Docker setup
- Vector similarity search with multiple distance functions
- Document chunking and metadata filtering
- Performance optimization guides and troubleshooting