pydantic_ai.models.openrouter

Setup

For details on how to set up authentication with this model, see model configuration for OpenRouter.

KnownOpenRouterProviders module-attribute

```
KnownOpenRouterProviders = Literal[
    "z-ai",
    "cerebras",
    "venice",
    "moonshotai",
    "morph",
    "stealth",
    "wandb",
    "klusterai",
    "openai",
    "sambanova",
    "amazon-bedrock",
    "mistral",
    "nextbit",
    "atoma",
    "ai21",
    "minimax",
    "baseten",
    "anthropic",
    "featherless",
    "groq",
    "lambda",
    "azure",
    "ncompass",
    "deepseek",
    "hyperbolic",
    "crusoe",
    "cohere",
    "mancer",
    "avian",
    "perplexity",
    "novita",
    "siliconflow",
    "switchpoint",
    "xai",
    "inflection",
    "fireworks",
```

OpenRouter

```
    "deepinfra",
    "inference-net",
    "inception",
    "atlas-cloud",
    "nvidia",
    "alibaba",
    "friendli",
    "infermatic",
    "targon",
    "ubicloud",
    "aion-labs",
    "liquid",
    "nineteen",
    "cloudflare",
    "nebius",
    "chutes",
    "enfer",
    "crofai",
    "open-inference",
    "phala",
    "gmicloud",
    "meta",
    "relace",
    "parasail",
    "together",
    "google-ai-studio",
    "google-vertex",
]
```

Known providers in the OpenRouter marketplace

OpenRouterProviderName `module-attribute`

```
OpenRouterProviderName = str | KnownOpenRouterProviders
```

Possible OpenRouter provider names.

Since OpenRouter is constantly updating their list of providers, we explicitly list some known providers but allow any name in the type hints. See the OpenRouter API for a full list.

OpenRouterTransforms `module-attribute`

```
OpenRouterTransforms = Literal['middle-out']
```

Available messages transforms for OpenRouter models with limited token windows.

Currently only supports 'middle-out', but is expected to grow in the future.

OpenRouterProviderConfig

Bases: `TypedDict`

Represents the 'Provider' object from the OpenRouter API.

Source code in `pydantic_ai_slim/pydantic_ai/models/openrouter.py`

```python
class OpenRouterProviderConfig(TypedDict, total=False):
    """Represents the 'Provider' object from the OpenRouter API."""

    order: list[OpenRouterProviderName]
    """List of provider slugs to try in order (e.g. ["anthropic", "openai"]). [See details](https://openrouter.ai/docs/features/provider-routing#ordering-specific-providers)"""

    allow_fallbacks: bool
    """Whether to allow backup providers when the primary is unavailable. [See details](https://openrouter.ai/docs/features/provider-routing#disabling-fallbacks)"""

    require_parameters: bool
    """Only use providers that support all parameters in your request."""

    data_collection: Literal['allow', 'deny']
    """Control whether to use providers that may store data. [See details](https://openrouter.ai/docs/features/provider-routing#requiring-providers-to-comply-with-data-policies)"""

    zdr: bool
    """Restrict routing to only ZDR (Zero Data Retention) endpoints. [See details](https://openrouter.ai/docs/features/provider-routing#zero-data-retention-enforcement)"""

    only: list[OpenRouterProviderName]
    """List of provider slugs to allow for this request. [See details](https://openrouter.ai/docs/features/provider-routing#allowing-only-specific-providers)"""

    ignore: list[str]
    """List of provider slugs to skip for this request. [See details](https://openrouter.ai/docs/features/provider-routing#ignoring-providers)"""

    quantizations: list[Literal['int4', 'int8', 'fp4', 'fp6', 'fp8', 'fp16', 'bf16', 'fp32', 'unknown']]
    """List of quantization levels to filter by (e.g. ["int4", "int8"]). [See details](https://openrouter.ai/docs/features/provider-routing#quantization)"""

    sort: Literal['price', 'throughput', 'latency']
    """Sort providers by price or throughput. (e.g. "price" or "throughput"). [See details](https://openrouter.ai/docs/features/provider-routing#provider-sorting)"""

    max_price: _OpenRouterMaxPrice
    """The maximum pricing you want to pay for this request. [See details](https://openrouter.ai/docs/features/provider-routing#max-price)"""
```

order `instance-attribute`

```
order: list[OpenRouterProviderName]
```

List of provider slugs to try in order (e.g. ["anthropic", "openai"]). See details

allow_fallbacks `instance-attribute`

```
allow_fallbacks: bool
```

Whether to allow backup providers when the primary is unavailable. See details

require_parameters `instance-attribute`

```
require_parameters: bool
```

Only use providers that support all parameters in your request.

data_collection `instance-attribute`

```
data_collection: Literal['allow', 'deny']
```

Control whether to use providers that may store data. See details

zdr `instance-attribute`

```
zdr: bool
```

Restrict routing to only ZDR (Zero Data Retention) endpoints. See details

only `instance-attribute`

```
only: list[OpenRouterProviderName]
```

List of provider slugs to allow for this request. See details

ignore `instance-attribute`

```
ignore: list[str]
```

List of provider slugs to skip for this request. See details

quantizations `instance-attribute`

```
quantizations: list[
    Literal[
        "int4",
```

```
        "int8",
        "fp4",
        "fp6",
        "fp8",
        "fp16",
        "bf16",
        "fp32",
        "unknown",
    ]
]
```

List of quantization levels to filter by (e.g. ["int4", "int8"]). See details

sort `instance-attribute`

```
sort: Literal['price', 'throughput', 'latency']
```

Sort providers by price or throughput. (e.g. "price" or "throughput"). See details

max_price `instance-attribute`

```
max_price: _OpenRouterMaxPrice
```

The maximum pricing you want to pay for this request. See details

OpenRouterReasoning

Bases: `TypedDict`

Configuration for reasoning tokens in OpenRouter requests.

Reasoning tokens allow models to show their step-by-step thinking process. You can configure this using either OpenAI-style effort levels or Anthropic-style token limits, but not both simultaneously.

Source code in `pydantic_ai_slim/pydantic_ai/models/openrouter.py`

```python
class OpenRouterReasoning(TypedDict, total=False):
    """Configuration for reasoning tokens in OpenRouter requests.

    Reasoning tokens allow models to show their step-by-step thinking process.
    You can configure this using either OpenAI-style effort levels or Anthropic-style
    token limits, but not both simultaneously.
    """

    effort: Literal['high', 'medium', 'low']
    """OpenAI-style reasoning effort level. Cannot be used with max_tokens."""

    max_tokens: int
    """Anthropic-style specific token limit for reasoning. Cannot be used with effort."""

    exclude: bool
    """Whether to exclude reasoning tokens from the response. Default is False. All models support this."""

    enabled: bool
    """Whether to enable reasoning with default parameters. Default is inferred from effort or max_tokens."""
```

effort `instance-attribute`

```python
effort: Literal['high', 'medium', 'low']
```

OpenAI-style reasoning effort level. Cannot be used with max_tokens.

max_tokens `instance-attribute`

```python
max_tokens: int
```

Anthropic-style specific token limit for reasoning. Cannot be used with effort.

exclude `instance-attribute`

```python
exclude: bool
```

Whether to exclude reasoning tokens from the response. Default is False. All models support this.

enabled `instance-attribute`

```python
enabled: bool
```

Whether to enable reasoning with default parameters. Default is inferred from effort or max_tokens.

OpenRouterUsageConfig

Bases: `TypedDict`

Configuration for OpenRouter usage.

Source code in `pydantic_ai_slim/pydantic_ai/models/openrouter.py`

```python
class OpenRouterUsageConfig(TypedDict, total=False):
    """Configuration for OpenRouter usage."""

    include: bool
```

OpenRouterModelSettings

Bases: `ModelSettings`

Settings used for an OpenRouter model request.

Source code in `pydantic_ai_slim/pydantic_ai/models/openrouter.py`

```python
class OpenRouterModelSettings(ModelSettings, total=False):
    """Settings used for an OpenRouter model request."""

    # ALL FIELDS MUST BE `openrouter_` PREFIXED SO YOU CAN MERGE THEM WITH OTHER MODELS.

    openrouter_models: list[str]
    """A list of fallback models.

    These models will be tried, in order, if the main model returns an error. [See details](https://openrouter.ai/docs/features/model-routing#the-models-parameter)
    """

    openrouter_provider: OpenRouterProviderConfig
    """OpenRouter routes requests to the best available providers for your model. By default, requests are load balanced across the top providers to maximize uptime.

    You can customize how your requests are routed using the provider object. [See more](https://openrouter.ai/docs/features/provider-routing)"""

    openrouter_preset: str
    """Presets allow you to separate your LLM configuration from your code.

    Create and manage presets through the OpenRouter web application to control provider routing, model selection, system prompts, and other parameters, then reference them in OpenRouter API requests. [See more](https://openrouter.ai/docs/features/presets)"""

    openrouter_transforms: list[OpenRouterTransforms]
    """To help with prompts that exceed the maximum context size of a model.

    Transforms work by removing or truncating messages from the middle of the prompt, until the prompt fits within the model's context window. [See more](https://openrouter.ai/docs/features/message-transforms)
    """

    openrouter_reasoning: OpenRouterReasoning
    """To control the reasoning tokens in the request.

    The reasoning config object consolidates settings for controlling reasoning strength across different models. [See more](https://openrouter.ai/docs/use-cases/reasoning-tokens)
    """

    openrouter_usage: OpenRouterUsageConfig
    """To control the usage of the model.

    The usage config object consolidates settings for enabling detailed usage information. [See more](https://openrouter.ai/docs/use-cases/usage-accounting)
    """
```

openrouter_models `instance-attribute`

```
openrouter_models: list[str]
```

A list of fallback models.

These models will be tried, in order, if the main model returns an error. See details

openrouter_provider `instance-attribute`

```
openrouter_provider: OpenRouterProviderConfig
```

OpenRouter routes requests to the best available providers for your model. By default, requests are load balanced across the top providers to maximize uptime.

You can customize how your requests are routed using the provider object. See more

openrouter_preset `instance-attribute`

```
openrouter_preset: str
```

Presets allow you to separate your LLM configuration from your code.

Create and manage presets through the OpenRouter web application to control provider routing, model selection, system prompts, and other parameters, then reference them in OpenRouter API requests. See more

openrouter_transforms `instance-attribute`

```
openrouter_transforms: list[OpenRouterTransforms]
```

To help with prompts that exceed the maximum context size of a model.

Transforms work by removing or truncating messages from the middle of the prompt, until the prompt fits within the model's context window. See more

openrouter_reasoning `instance-attribute`

```
openrouter_reasoning: OpenRouterReasoning
```

To control the reasoning tokens in the request.

The reasoning config object consolidates settings for controlling reasoning strength across different models. See more

openrouter_usage `instance-attribute`

```
openrouter_usage: OpenRouterUsageConfig
```

To control the usage of the model.

The usage config object consolidates settings for enabling detailed usage information. See more

OpenRouterModel

Bases: `OpenAIChatModel`

Extends OpenAIModel to capture extra metadata for Openrouter.

Source code in `pydantic_ai_slim/pydantic_ai/models/openrouter.py`

```python
class OpenRouterModel(OpenAIChatModel):
    """Extends OpenAIModel to capture extra metadata for Openrouter."""

    def __init__(
        self,
        model_name: str,
        *,
        provider: Literal['openrouter'] | Provider[AsyncOpenAI] =
'openrouter',
        profile: ModelProfileSpec | None = None,
        settings: ModelSettings | None = None,
    ):
        """Initialize an OpenRouter model.

        Args:
            model_name: The name of the model to use.
            provider: The provider to use for authentication and API access.
If not provided, a new provider will be created with the default settings.
            profile: The model profile to use. Defaults to a profile picked
by the provider based on the model name.
            settings: Model-specific settings that will be used as defaults
for this model.
        """
        super().__init__(model_name, provider=provider or
OpenRouterProvider(), profile=profile, settings=settings)

    @override
    def prepare_request(
        self,
        model_settings: ModelSettings | None,
        model_request_parameters: ModelRequestParameters,
    ) -> tuple[ModelSettings | None, ModelRequestParameters]:
        merged_settings, customized_parameters =
super().prepare_request(model_settings, model_request_parameters)
        new_settings =
_openrouter_settings_to_openai_settings(cast(OpenRouterModelSettings,
merged_settings or {}))
        return new_settings, customized_parameters

    @override
    def _validate_completion(self, response: chat.ChatCompletion) ->
_OpenRouterChatCompletion:
        response =
_OpenRouterChatCompletion.model_validate(response.model_dump())

        if error := response.error:
            raise ModelHTTPError(status_code=error.code,
model_name=response.model, body=error.message)

        return response

    @override
    def _process_thinking(self, message: chat.ChatCompletionMessage) ->
list[ThinkingPart] | None:
        assert isinstance(message, _OpenRouterCompletionMessage)

        if reasoning_details := message.reasoning_details:
```

```
            return [_from_reasoning_detail(detail) for detail in
reasoning_details]
        else:
            return super()._process_thinking(message)

    @override
    def _process_provider_details(self, response: chat.ChatCompletion) ->
dict[str, Any] | None:
        assert isinstance(response, _OpenRouterChatCompletion)

        provider_details = super()._process_provider_details(response) or {}
        provider_details.update(_map_openrouter_provider_details(response))
        return provider_details or None

    @dataclass
    class _MapModelResponseContext(OpenAIChatModel._MapModelResponseContext):
# type: ignore[reportPrivateUsage]
        reasoning_details: list[dict[str, Any]] =
field(default_factory=list[dict[str, Any]])

        def _into_message_param(self) ->
chat.ChatCompletionAssistantMessageParam:
            message_param = super()._into_message_param()
            if self.reasoning_details:
                message_param['reasoning_details'] = self.reasoning_details
# type: ignore[reportGeneralTypeIssues]
            return message_param

        @override
        def _map_response_thinking_part(self, item: ThinkingPart) -> None:
            assert isinstance(self._model, OpenRouterModel)
            if item.provider_name == self._model.system:
                if reasoning_detail := _into_reasoning_detail(item):  #
pragma: lax no cover

self.reasoning_details.append(reasoning_detail.model_dump())
            else:  # pragma: lax no cover
                super()._map_response_thinking_part(item)

    @property
    @override
    def _streamed_response_cls(self):
        return OpenRouterStreamedResponse

    @override
    def _map_finish_reason(  # type: ignore[reportIncompatibleMethodOverride]
        self, key: Literal['stop', 'length', 'tool_calls', 'content_filter',
'error']
    ) -> FinishReason | None:
        return _CHAT_FINISH_REASON_MAP.get(key)
```

__init__

```
__init__(
    model_name: str,
    *,
    provider: (
        Literal["openrouter"] | Provider[AsyncOpenAI]
    ) = "openrouter",
```

```
    profile: ModelProfileSpec | None = None,
    settings: ModelSettings | None = None
)
```

Initialize an OpenRouter model.

Parameters:

Name	Type	Description	Default
`model_name`	`str`	The name of the model to use.	*required*
`provider`	`Literal['openrouter'] | Provider[AsyncOpenAI]`	The provider to use for authentication and API access. If not provided, a new provider will be created with the default settings.	`'openrouter'`
`profile`	`ModelProfileSpec | None`	The model profile to use. Defaults to a profile picked by the provider based on the model name.	`None`
`settings`	`ModelSettings | None`	Model-specific settings that will be used as defaults for this model.	`None`

Source code in `pydantic_ai_slim/pydantic_ai/models/openrouter.py`

```
522  def __init__(
523      self,
524      model_name: str,
525      *,
526      provider: Literal['openrouter'] | Provider[AsyncOpenAI] = 'openrouter',
527      profile: ModelProfileSpec | None = None,
528      settings: ModelSettings | None = None,
529  ):
530      """Initialize an OpenRouter model.
531
532      Args:
533          model_name: The name of the model to use.
534          provider: The provider to use for authentication and API access. If
535  not provided, a new provider will be created with the default settings.
536          profile: The model profile to use. Defaults to a profile picked by
537  the provider based on the model name.
538          settings: Model-specific settings that will be used as defaults for
      this model.
          """
          super().__init__(model_name, provider=provider or OpenRouterProvider(),
      profile=profile, settings=settings)
```

OpenRouterStreamedResponse `dataclass`

Bases: `OpenAIStreamedResponse`

Implementation of `StreamedResponse` for OpenRouter models.

Source code in `pydantic_ai_slim/pydantic_ai/models/openrouter.py`

```python
@dataclass
class OpenRouterStreamedResponse(OpenAIStreamedResponse):
    """Implementation of `StreamedResponse` for OpenRouter models."""

    @override
    async def _validate_response(self):
        try:
            async for chunk in self._response:
                yield _OpenRouterChatCompletionChunk.model_validate(chunk.model_dump())
        except APIError as e:
            error = _OpenRouterError.model_validate(e.body)
            raise ModelHTTPError(status_code=error.code, model_name=self._model_name, body=error.message)

    @override
    def _map_thinking_delta(self, choice: chat_completion_chunk.Choice) -> Iterable[ModelResponseStreamEvent]:
        assert isinstance(choice, _OpenRouterChunkChoice)

        if reasoning_details := choice.delta.reasoning_details:
            for i, detail in enumerate(reasoning_details):
                thinking_part = _from_reasoning_detail(detail)
                # Use unique vendor_part_id for each reasoning detail type to prevent
                # different detail types (e.g., reasoning.text, reasoning.encrypted)
                # from being incorrectly merged into a single ThinkingPart.
                # This is required for Gemini 3 Pro which returns multiple reasoning
                # detail types that must be preserved separately for thought_signature handling.
                vendor_id = f'reasoning_detail_{detail.type}_{i}'
                yield from self._parts_manager.handle_thinking_delta(
                    vendor_part_id=vendor_id,
                    id=thinking_part.id,
                    content=thinking_part.content,
                    signature=thinking_part.signature,
                    provider_name=self._provider_name,
                    provider_details=thinking_part.provider_details,
                )
        else:
            return super()._map_thinking_delta(choice)

    @override
    def _map_provider_details(self, chunk: chat.ChatCompletionChunk) -> dict[str, Any] | None:
        assert isinstance(chunk, _OpenRouterChatCompletionChunk)

        provider_details = super()._map_provider_details(chunk) or {}
        provider_details.update(_map_openrouter_provider_details(chunk))
        return provider_details or None

    @override
    def _map_finish_reason(  # type: ignore[reportIncompatibleMethodOverride]
        self, key: Literal['stop', 'length', 'tool_calls', 'content_filter', 'error']
```

```
            ) -> FinishReason | None:
                return _CHAT_FINISH_REASON_MAP.get(key)
```