`pydantic_ai.models.gemini`

Custom interface to the `generativelanguage.googleapis.com` API using HTTPX and [Pydantic](https://docs.pydantic.dev/latest/.

The Google SDK for interacting with the `generativelanguage.googleapis.com` API `google-generativeai` reads like it was written by a Java developer who thought they knew everything about OOP, spent 30 minutes trying to learn Python, gave up and decided to build the library to prove how horrible Python is. It also doesn't use httpx for HTTP requests, and tries to implement tool calling itself, but doesn't use Pydantic or equivalent for validation.

We therefore implement support for the API directly.

Despite these shortcomings, the Gemini model is actually quite powerful and very fast.

Setup

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

GeminiModelName `module-attribute`

```
GeminiModelName = Literal[
    "gemini-1.5-flash",
    "gemini-1.5-flash-8b",
    "gemini-1.5-pro",
    "gemini-1.0-pro",
]
```

Named Gemini models.

See the Gemini API docs for a full list.

GeminiModel `dataclass`

Bases: `Model`

A model that uses Gemini via `generativelanguage.googleapis.com` API.

This is implemented from scratch rather than using a dedicated SDK, good API documentation is available here.

Apart from `__init__`, all methods are private or match those of the base class.

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

```
@dataclass(init=False)
class GeminiModel(Model):
    """A model that uses Gemini via `generativelanguage.googleapis.com` API.

    This is implemented from scratch rather than using a dedicated SDK, good API documentation is
    available [here](https://ai.google.dev/api).

    Apart from `__init__`, all methods are private or match those of the base class.
    """

    model_name: GeminiModelName
    auth: AuthProtocol
    http_client: AsyncHTTPClient
    url: str

    def __init__(
        self,
        model_name: GeminiModelName,
        *,
        api_key: str | None = None,
        http_client: AsyncHTTPClient | None = None,
        url_template: str = 'https://generativelanguage.googleapis.com/v1beta/models/{model}:',
    ):
        """Initialize a Gemini model.

        Args:
            model_name: The name of the model to use.
            api_key: The API key to use for authentication, if not provided, the `GEMINI_API_KEY` environment variable
                will be used if available.
            http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
            url_template: The URL template to use for making requests, you shouldn't need to change this,
                docs [here](https://ai.google.dev/gemini-api/docs/quickstart?lang=rest#make-first-request),
                `model` is substituted with the model name, and `function` is added to the end of the URL.
        """
        self.model_name = model_name
        if api_key is None:
            if env_api_key := os.getenv('GEMINI_API_KEY'):
                api_key = env_api_key
            else:
                raise exceptions.UserError('API key must be provided or set in the GEMINI_API_KEY environment variable')
        self.auth = ApiKeyAuth(api_key)
        self.http_client = http_client or cached_async_http_client()
        self.url = url_template.format(model=model_name)

    async def agent_model(
        self,
        *,
        function_tools: list[ToolDefinition],
        allow_text_result: bool,
        result_tools: list[ToolDefinition],
    ) -> GeminiAgentModel:
        return GeminiAgentModel(
            http_client=self.http_client,
            model_name=self.model_name,
            auth=self.auth,
            url=self.url,
            function_tools=function_tools,
            allow_text_result=allow_text_result,
            result_tools=result_tools,
        )

    def name(self) -> str:
        return self.model_name
```

__init__

```
__init__(
    model_name: GeminiModelName,
    *,
    api_key: str | None = None,
    http_client: AsyncClient | None = None,
    url_template: str = "https://generativelanguage.googleapis.com/v1beta/models/{model}:"
)
```

Initialize a Gemini model.

Parameters:

Name	Type	Description	Default
`model_name`	`GeminiModelName`	The name of the model to use.	*required*
`api_key`	`str | None`	The API key to use for authentication, if not provided, the `GEMINI_API_KEY` environment variable will be used if available.	`None`
`http_client`	`AsyncClient | None`	An existing `httpx.AsyncClient` to use for making HTTP requests.	`None`
`url_template`	`str`	The URL template to use for making requests, you shouldn't need to change this, docs here, `model` is substituted with the model name, and `function` is added to the end of the URL.	`'https://generativelanguage.googleapis.com/v1beta/models/{model}:'`

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

```python
def __init__(
    self,
    model_name: GeminiModelName,
    *,
    api_key: str | None = None,
    http_client: AsyncHTTPClient | None = None,
    url_template: str = 'https://generativelanguage.googleapis.com/v1beta/models/{model}:',
):
    """Initialize a Gemini model.

    Args:
        model_name: The name of the model to use.
        api_key: The API key to use for authentication, if not provided, the `GEMINI_API_KEY` environment variable
            will be used if available.
        http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
        url_template: The URL template to use for making requests, you shouldn't need to change this,
            docs [here](https://ai.google.dev/gemini-api/docs/quickstart?lang=rest#make-first-request),
            `model` is substituted with the model name, and `function` is added to the end of the URL.
    """
    self.model_name = model_name
    if api_key is None:
        if env_api_key := os.getenv('GEMINI_API_KEY'):
            api_key = env_api_key
        else:
            raise exceptions.UserError('API key must be provided or set in the GEMINI_API_KEY environment variable')
    self.auth = ApiKeyAuth(api_key)
    self.http_client = http_client or cached_async_http_client()
    self.url = url_template.format(model=model_name)
```

AuthProtocol

Bases: `Protocol`

Abstract definition for Gemini authentication.

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

```python
class AuthProtocol(Protocol):
    """Abstract definition for Gemini authentication."""

    async def headers(self) -> dict[str, str]: ...
```

ApiKeyAuth `dataclass`

Authentication using an API key for the `X-Goog-Api-Key` header.

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

```python
@dataclass
class ApiKeyAuth:
    """Authentication using an API key for the `X-Goog-Api-Key` header."""

    api_key: str

    async def headers(self) -> dict[str, str]:
        # https://cloud.google.com/docs/authentication/api-keys-use#using-with-rest
        return {'X-Goog-Api-Key': self.api_key}
```

GeminiAgentModel `dataclass`

Bases: `AgentModel`

Implementation of `AgentModel` for Gemini models.

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

```python
@dataclass(init=False)
class GeminiAgentModel(AgentModel):
    """Implementation of `AgentModel` for Gemini models."""

    http_client: AsyncHTTPClient
    model_name: GeminiModelName
    auth: AuthProtocol
    tools: _GeminiTools | None
    tool_config: _GeminiToolConfig | None
    url: str

    def __init__(
        self,
        http_client: AsyncHTTPClient,
        model_name: GeminiModelName,
        auth: AuthProtocol,
        url: str,
        function_tools: list[ToolDefinition],
        allow_text_result: bool,
        result_tools: list[ToolDefinition],
    ):
        check_allow_model_requests()
        tools = [_function_from_abstract_tool(t) for t in function_tools]
        if result_tools:
            tools += [_function_from_abstract_tool(t) for t in result_tools]

        if allow_text_result:
            tool_config = None
        else:
            tool_config = _tool_config([t['name'] for t in tools])

        self.http_client = http_client
        self.model_name = model_name
        self.auth = auth
        self.tools = _GeminiTools(function_declarations=tools) if tools else None
        self.tool_config = tool_config
        self.url = url

    async def request(self, messages: list[Message]) -> tuple[ModelAnyResponse, result.Cost]:
        async with self._make_request(messages, False) as http_response:
            response = _gemini_response_ta.validate_json(await http_response.aread())
        return self._process_response(response), _metadata_as_cost(response)

    @asynccontextmanager
    async def request_stream(self, messages: list[Message]) -> AsyncIterator[EitherStreamedResponse]:
        async with self._make_request(messages, True) as http_response:
            yield await self._process_streamed_response(http_response)

    @asynccontextmanager
    async def _make_request(self, messages: list[Message], streamed: bool) -> AsyncIterator[HTTPResponse]:
        contents: list[_GeminiContent] = []
        sys_prompt_parts: list[_GeminiTextPart] = []
        for m in messages:
            either_content = self._message_to_gemini(m)
            if left := either_content.left:
                sys_prompt_parts.append(left.value)
            else:
                contents.append(either_content.right)

        request_data = _GeminiRequest(contents=contents)
        if sys_prompt_parts:
            request_data['system_instruction'] = _GeminiTextContent(role='user', parts=sys_prompt_parts)
        if self.tools is not None:
            request_data['tools'] = self.tools
        if self.tool_config is not None:
            request_data['tool_config'] = self.tool_config

        url = self.url + ('streamGenerateContent' if streamed else 'generateContent')

        headers = {
            'Content-Type': 'application/json',
            'User-Agent': get_user_agent(),
            **await self.auth.headers(),
        }

        request_json = _gemini_request_ta.dump_json(request_data, by_alias=True)

        async with self.http_client.stream('POST', url, content=request_json, headers=headers) as r:
            if r.status_code != 200:
                await r.aread()
                raise exceptions.UnexpectedModelBehavior(f'Unexpected response from gemini {r.status_code}', r.text)
            yield r

    @staticmethod
    def _process_response(response: _GeminiResponse) -> ModelAnyResponse:
        either = _extract_response_parts(response)
        if left := either.left:
            return _structured_response_from_parts(left.value)
        else:
            return ModelTextResponse(content=''.join(part['text'] for part in either.right))

    @staticmethod
    async def _process_streamed_response(http_response: HTTPResponse) -> EitherStreamedResponse:
        """Process a streamed response, and prepare a streaming response to return."""
        aiter_bytes = http_response.aiter_bytes()
        start_response: _GeminiResponse | None = None
        content = bytearray()

        async for chunk in aiter_bytes:
            content.extend(chunk)
            responses = _gemini_streamed_response_ta.validate_json(
                content,
                experimental_allow_partial='trailing-strings',
            )
            if responses:
                last = responses[-1]
                if last['candidates'] and last['candidates'][0]['content']['parts']:
                    start_response = last
                    break

        if start_response is None:
            raise UnexpectedModelBehavior('Streamed response ended without content or tool calls')

        if _extract_response_parts(start_response).is_left():
            return GeminiStreamStructuredResponse(_content=content, _stream=aiter_bytes)
        else:
            return GeminiStreamTextResponse(_json_content=content, _stream=aiter_bytes)

    @staticmethod
    def _message_to_gemini(m: Message) -> _utils.Either[_GeminiTextPart, _GeminiContent]:
        """Convert a message to a _GeminiTextPart for "system_instructions" or _GeminiContent for "contents"."""
        if m.role == 'system':
            # SystemPrompt ->
            return _utils.Either(left=_GeminiTextPart(text=m.content))
        elif m.role == 'user':
            # UserPrompt ->
            return _utils.Either(right=_content_user_text(m.content))
        elif m.role == 'tool-return':
            # ToolReturn ->
            return _utils.Either(right=_content_function_return(m))
        elif m.role == 'retry-prompt':
            # RetryPrompt ->
            return _utils.Either(right=_content_function_retry(m))
        elif m.role == 'model-text-response':
            # ModelTextResponse ->
            return _utils.Either(right=_content_model_text(m.content))
        elif m.role == 'model-structured-response':
            # ModelStructuredResponse ->
            return _utils.Either(right=_content_function_call(m))
        else:
            assert_never(m)
```

GeminiStreamTextResponse `dataclass`

Bases: `StreamTextResponse`

Implementation of `StreamTextResponse` for the Gemini model.

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

```python
@dataclass
class GeminiStreamTextResponse(StreamTextResponse):
    """Implementation of `StreamTextResponse` for the Gemini model."""

    _json_content: bytearray
    _stream: AsyncIterator[bytes]
    _position: int = 0
    _timestamp: datetime = field(default_factory=_utils.now_utc, init=False)
    _cost: result.Cost = field(default_factory=result.Cost, init=False)

    async def __anext__(self) -> None:
        chunk = await self._stream.__anext__()
        self._json_content.extend(chunk)

    def get(self, *, final: bool = False) -> Iterable[str]:
        if final:
            all_items = pydantic_core.from_json(self._json_content)
            new_items = all_items[self._position :]
            self._position = len(all_items)
            new_responses = _gemini_streamed_response_ta.validate_python(new_items)
        else:
            all_items = pydantic_core.from_json(self._json_content, allow_partial=True)
            new_items = all_items[self._position : -1]
            self._position = len(all_items) - 1
            new_responses = _gemini_streamed_response_ta.validate_python(
                new_items, experimental_allow_partial='trailing-strings'
            )
        for r in new_responses:
            self._cost += _metadata_as_cost(r)
            parts = r['candidates'][0]['content']['parts']
            if _all_text_parts(parts):
                for part in parts:
                    yield part['text']
            else:
                raise UnexpectedModelBehavior(
                    'Streamed response with unexpected content, expected all parts to be text'
                )

    def cost(self) -> result.Cost:
        return self._cost

    def timestamp(self) -> datetime:
        return self._timestamp
```

GeminiStreamStructuredResponse `dataclass`

Bases: `StreamStructuredResponse`

Implementation of `StreamStructuredResponse` for the Gemini model.

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

```python
@dataclass
class GeminiStreamStructuredResponse(StreamStructuredResponse):
    """Implementation of `StreamStructuredResponse` for the Gemini model."""

    _content: bytearray
    _stream: AsyncIterator[bytes]
    _timestamp: datetime = field(default_factory=_utils.now_utc, init=False)
    _cost: result.Cost = field(default_factory=result.Cost, init=False)

    async def __anext__(self) -> None:
        chunk = await self._stream.__anext__()
        self._content.extend(chunk)

    def get(self, *, final: bool = False) -> ModelStructuredResponse:
        """Get the `ModelStructuredResponse` at this point.

        NOTE: It's not clear how the stream of responses should be combined because Gemini seems to always
        reply with a single response, when returning a structured data.

        I'm therefore assuming that each part contains a complete tool call, and not trying to combine data from
        separate parts.
        """
        responses = _gemini_streamed_response_ta.validate_json(
            self._content,
            experimental_allow_partial='off' if final else 'trailing-strings',
        )
        combined_parts: list[_GeminiFunctionCallPart] = []
        self._cost = result.Cost()
        for r in responses:
            self._cost += _metadata_as_cost(r)
            candidate = r['candidates'][0]
            parts = candidate['content']['parts']
            if _all_function_call_parts(parts):
                combined_parts.extend(parts)
            elif not candidate.get('finish_reason'):
                # you can get an empty text part along with the finish_reason, so we ignore that case
                raise UnexpectedModelBehavior(
                    'Streamed response with unexpected content, expected all parts to be function calls'
                )
        return _structured_response_from_parts(combined_parts, timestamp=self._timestamp)

    def cost(self) -> result.Cost:
        return self._cost

    def timestamp(self) -> datetime:
        return self._timestamp
```

get

```python
get(*, final: bool = False) -> ModelStructuredResponse
```

Get the `ModelStructuredResponse` at this point.

NOTE: It's not clear how the stream of responses should be combined because Gemini seems to always reply with a single response, when returning a structured data.

I'm therefore assuming that each part contains a complete tool call, and not trying to combine data from separate parts.

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

```python
def get(self, *, final: bool = False) -> ModelStructuredResponse:
    """Get the `ModelStructuredResponse` at this point.

    NOTE: It's not clear how the stream of responses should be combined because Gemini seems to always
    reply with a single response, when returning a structured data.

    I'm therefore assuming that each part contains a complete tool call, and not trying to combine data from
    separate parts.
    """
    responses = _gemini_streamed_response_ta.validate_json(
        self._content,
        experimental_allow_partial='off' if final else 'trailing-strings',
    )
    combined_parts: list[_GeminiFunctionCallPart] = []
    self._cost = result.Cost()
    for r in responses:
        self._cost += _metadata_as_cost(r)
        candidate = r['candidates'][0]
        parts = candidate['content']['parts']
        if _all_function_call_parts(parts):
            combined_parts.extend(parts)
        elif not candidate.get('finish_reason'):
            # you can get an empty text part along with the finish_reason, so we ignore that case
            raise UnexpectedModelBehavior(
                'Streamed response with unexpected content, expected all parts to be function calls'
            )
    return _structured_response_from_parts(combined_parts, timestamp=self._timestamp)
```