Add icon Add cover

Unified AI Abstraction Layer

The Problem We're Facing

1.1 Where We Are Today

We've built three AI-powered products — **DocsAI**, **InsightsAI**, and **EngageAI** — and each has grown organically with its own AI integration. Today, when you look at our codebase, here's what you'll find:

In DocsAI (`llm_service.py` - 2,700+ lines):

Python

```python
# We have GPT-5 specific workarounds scattered throughout
def _prepare_token_params(self, deployment: str, max_tokens: int):
    if deployment.lower().startswith('gpt-5'):
        # GPT-5 uses 'max_completion_tokens' instead of 'max_tokens'
        # Also needs minimum 6000 tokens for reasoning
        adjusted_tokens = max(max_tokens, 6000)
        return {"max_completion_tokens": adjusted_tokens}
    else:
        return {"max_tokens": max_tokens}

```

This pattern repeats everywhere — temperature handling, schema strict mode, message formatting. Every time OpenAI changes something, we're hunting through thousands of lines of code.

1.2 The Pain Points (Real Examples from Our Codebase)

	Problem	What's Happening	Business Impact
1	**Model-Specific Code Explosion**	GPT-5 needs `max_completion_tokens`, GPT-4 uses `max_tokens`. GPT-5 doesn't support custom temperature. We have 15+ `if model == 'gpt-5'` checks.	Every new model release = 2-3 days of hunting and fixing
2	**Copy-Paste Integration**	InsightsAI copied DocsAI's LLM code. EngageAI did the same. Now we have 3 versions diverging.	Bug fixes don't propagate. Security patches are nightmares.
3	**Can't Switch Models**	Customer asks: "Can we use Claude instead of GPT-4o for extraction?" Answer: "That's a 3-week project."	Lost deals, frustrated customers
4	**No Anthropic/Gemini Support**	We're locked to Azure OpenAI. Competitors are offering Claude for chat. We can't.	Competitive disadvantage
5	**Streaming is Missing**	EngageAI needs real-time chat streaming. Our current architecture doesn't support it.	Poor UX in chat applications

1.3 What Triggered This Initiative

Three things happened in the last quarter that made this urgent:

1. **GPT-5 Launch Broke Production** — OpenAI changed parameter names. We spent 4 days patching three products.
2. **Enterprise Customer Request** — A Fortune 500 prospect requires Claude (Anthropic) for compliance reasons. We had to decline.
3. **EngageAI Streaming Requirement** — The product team needs streaming chat. Our architecture can't support it without major rework.

1.4 The Models We Need to Support

We're not just talking about GPT-4. Here's the full landscape:

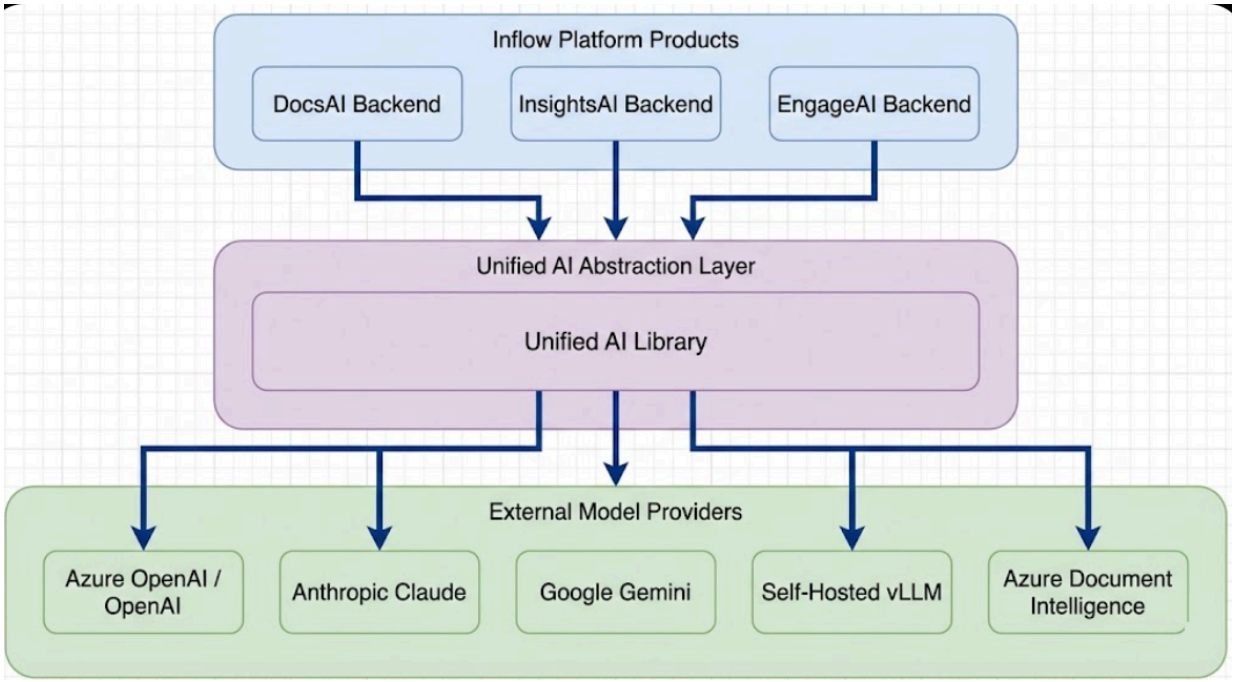
	Provider	Models We Need	Use Case
1	**OpenAI/Azure OpenAI**	GPT-4o (nano/mini), GPT-4.1, GPT-5, GPT-5.1, GPT-5.2, o1, o3, o3 Pro	Primary extraction, reasoning
2	**Anthropic**	Claude Sonnet 4.5, Claude Opus 4.5, Claude Haiku 4.5	Enterprise customers, long context
3	**Google**	Gemini 3 Flash, Gemini 3 Pro, Gemini 2.5 Pro	Cost optimization, multimodal
4	**vLLM/Self-hosted**	Mistral 2, Mistral 3, DeepSeek OCR	Air-gapped deployments, specialized OCR
5	**Specialized**	Azure Document Intelligence	Document layout analysis

That's **15+ models across 5 providers** — and we need to handle all of them with a single, clean interface.

2. Our Proposed Solution

2.1 The Core Idea

We're proposing a **Unified AI Abstraction Layer** — a shared Python library called `inflow-ai-core` that sits between our products and AI providers.



The simple version:

```python
# Before (DocsAI today - provider-specific, messy)
from openai import AzureOpenAI
client = AzureOpenAI(endpoint=..., api_key=...)
response = client.chat.completions.create(
    model="gpt-4o",
    messages=[...],
    max_tokens=2000,  # Wait, is it max_tokens or max_completion_tokens for GPT-5?
    temperature=0.3,  # GPT-5 doesn't support this!
    response_format={"type": "json_schema", "json_schema": {...}}  # Different per model!
)

# After (with our abstraction layer)
from inflow_ai_core import AIClient

client = AIClient()
response = await client.generate(
    model="gpt-4o",  # Or "claude-sonnet-4.5" or "gemini-3-pro" — same API
    messages=[...],
    temperature=0.3,
    max_tokens=2000,
    schema=my_extraction_schema  # We handle provider-specific formatting
)

```

The library handles all the messy provider-specific stuff internally. Our products just say "give me a response from this model" and it works.

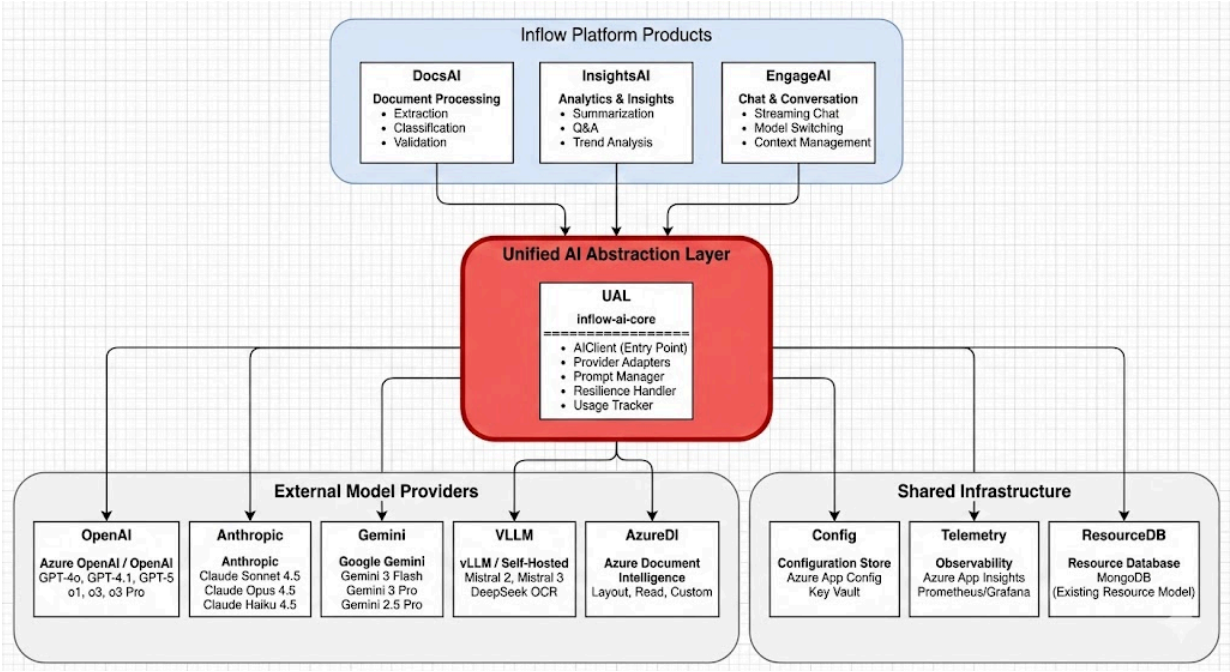
2.2 Why a Library, Not a Microservice?

We considered building this as a separate AI Gateway microservice. Here's why we chose a library instead:

	Approach	Pros	Cons
1	**Microservice**	Central management, single deployment	Network latency (+50-100ms), new infrastructure, operational overhead, single point of failure
2	**Shared Library ✓**	Zero latency, scales with each product, no new infra	Need to coordinate library updates across products

Our decision: Library approach. The latency cost of a microservice is unacceptable for real-time chat (EngageAI) and high-volume document processing (DocsAI). We'll use semantic versioning and maintain backward compatibility.

2.3 How It Fits Into Our Architecture



Key point: We're not throwing away our existing Resource database model. The library will read from the same MongoDB collection we use today — just through a cleaner abstraction.

---

3. How We're Going to Build It

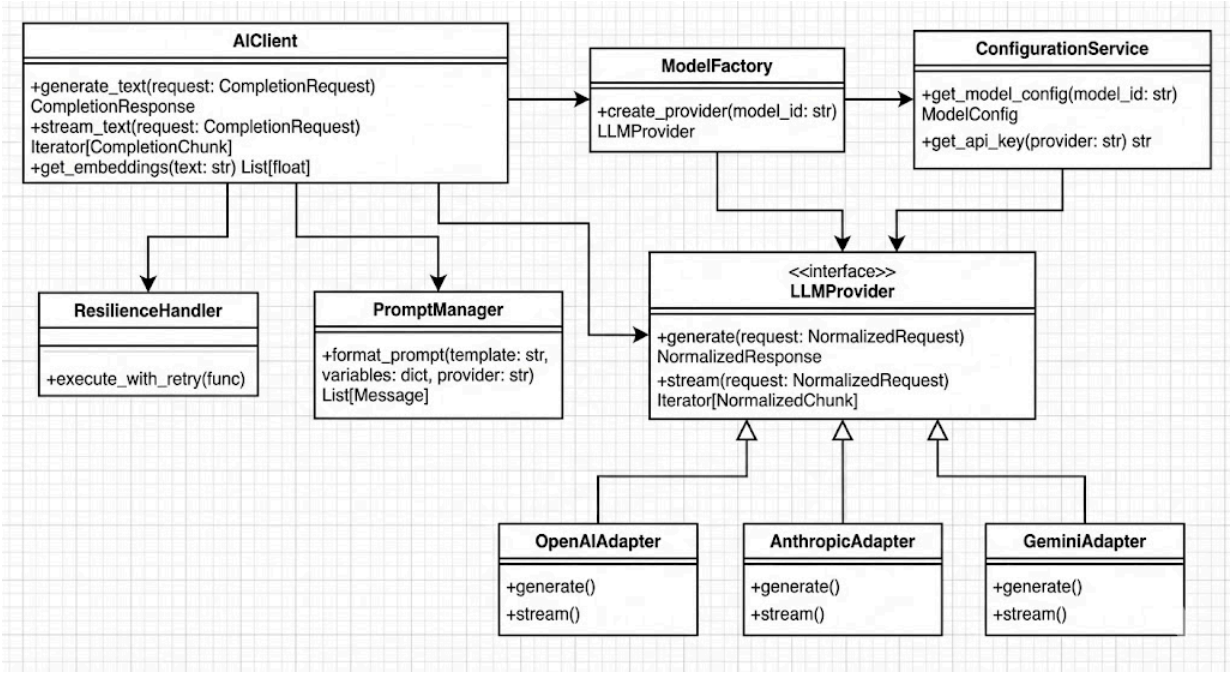
3.1 The Design Pattern: Adapter + Factory

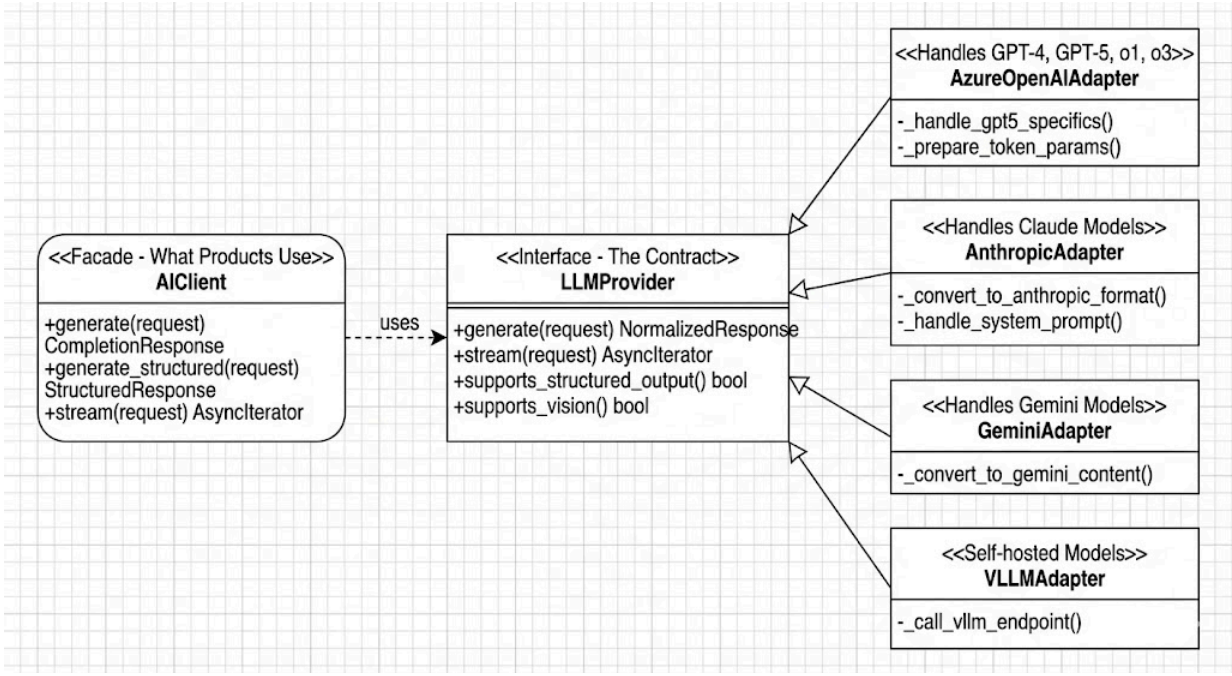
We're using a well-established pattern here. The idea is simple:

1. **Every provider gets its own Adapter** — OpenAI has one, Anthropic has one, Gemini has one
2. **All Adapters implement the same interface** — `generate()`, `stream()`, `supports_vision()`, etc.
3. **A Factory picks the right Adapter** — You say "gpt-4o", it gives you the OpenAI adapter

This means when Anthropic changes their API, we fix ONE file. When we add a new provider, we write ONE new adapter. No ripple effects.

TODO: facade pattern





3.2 What Each Component Does

Let me explain the components and why we need each one:

	Component	What It Does	Why We Need It
1	**AIClient**	The only thing products talk to. Simple API: `generate()`, `stream()`, `generate_structured()`	Products shouldn't know about providers. One interface to learn.
2	**ModelFactory**	Maps model names to adapters. "gpt-4o" → AzureOpenAIAdapter	New model support = add one line to the registry
3	**Provider Adapters**	Provider-specific code. Each handles its quirks internally.	Isolates messy API differences. OpenAI change? Fix one file.
4	**PromptManager**	Formats prompts for each provider. Handles vision content, schema injection.	Anthropic wants system prompt separate. Gemini has different image format. We abstract this.
5	**ResilienceHandler**	Retries, circuit breakers, fallbacks	"If GPT-4o fails, try GPT-4o-mini" — automatic
6	**UsageTracker**	Tracks tokens, calculates costs, logs to DB	Billing, analytics, cost optimization
7	**ConfigurationService**	Reads from our existing Resource database	We don't reinvent config. Use what we have.

3.3 The Package Structure

Here's how the code will be organized:

Python

```
inflow-ai-core/
├── pyproject.toml                    # Package metadata, dependencies
├── src/
│   └── inflow_ai_core/
│       ├── __init__.py               # Public API exports
│       ├── client.py                 # AIClient - the main entry point
│       │
│       ├── models/                   # Data structures (Pydantic)
│       │   ├── requests.py           # CompletionRequest, StructuredRequest
│       │   ├── responses.py          # CompletionResponse, CompletionChunk
│       │   └── config.py             # ModelConfig, RetryConfig
│       │
│       ├── providers/                # One file per provider
│       │   ├── base.py               # LLMProvider abstract class
│       │   ├── azure_openai.py       # AzureOpenAIAdapter (GPT-4, GPT-5, o1, o3)
│       │   ├── anthropic.py          # AnthropicAdapter (Claude family)
│       │   ├── gemini.py             # GeminiAdapter
│       │   ├── vllm.py               # VLLMAdapter (Mistral, DeepSeek)
│       │   └── factory.py            # ModelFactory
│       │
│       ├── prompts/
│       │   └── manager.py            # PromptManager - format conversion
│       │
│       ├── resilience/
│       │   ├── retry.py              # Exponential backoff, jitter
│       │   └── circuit_breaker.py    # Fail-fast when provider is down
│       │
│       └── tracking/
│           └── usage.py              # Token tracking, cost calculation
│
└── tests/                            # Unit & integration tests

```

Why this structure?

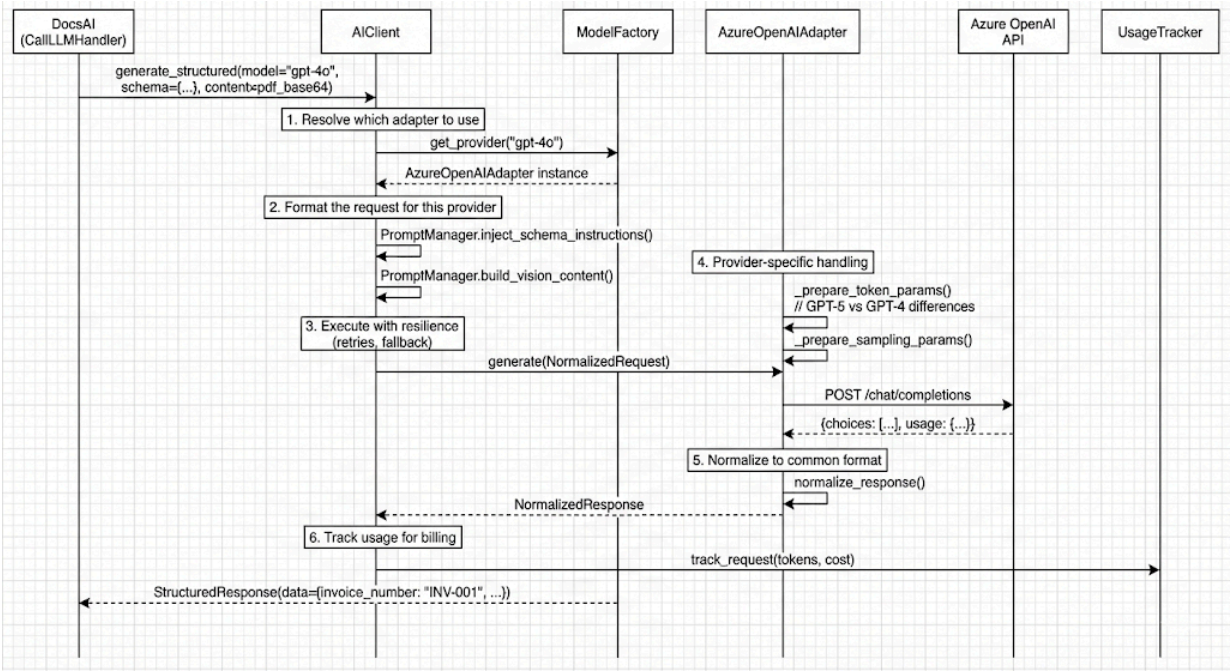
- `providers/` — When we add Cohere tomorrow, we add `cohere.py`. Nothing else changes.
- `models/` — Type-safe Pydantic models. No more "what fields does this dict have?"
- `resilience/` — Centralized. Not copy-pasted into each product.

4. How Data Flows Through the System

Let me walk you through what happens when our products call the library.

4.1 Standard Request Flow (DocsAI Extraction)

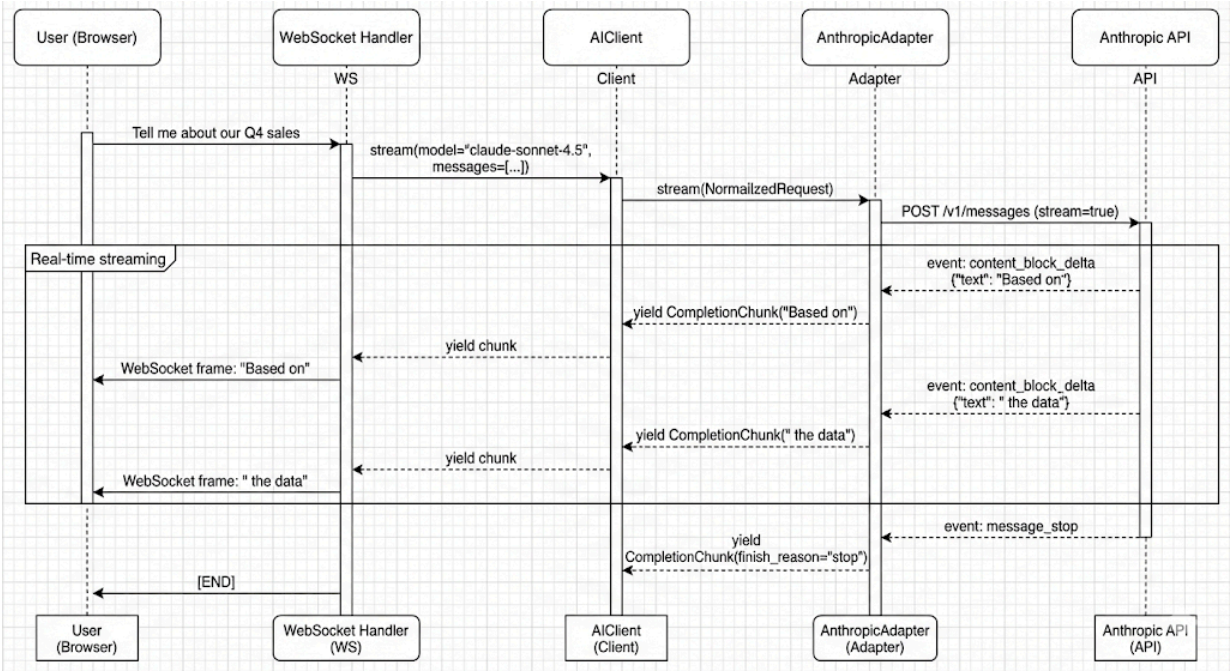
When DocsAI calls the library to extract invoice data:



The key insight: DocsAI's code doesn't change when we switch from GPT-4o to Claude. The adapter handles everything.

4.2 Streaming Flow (EngageAI Chat)

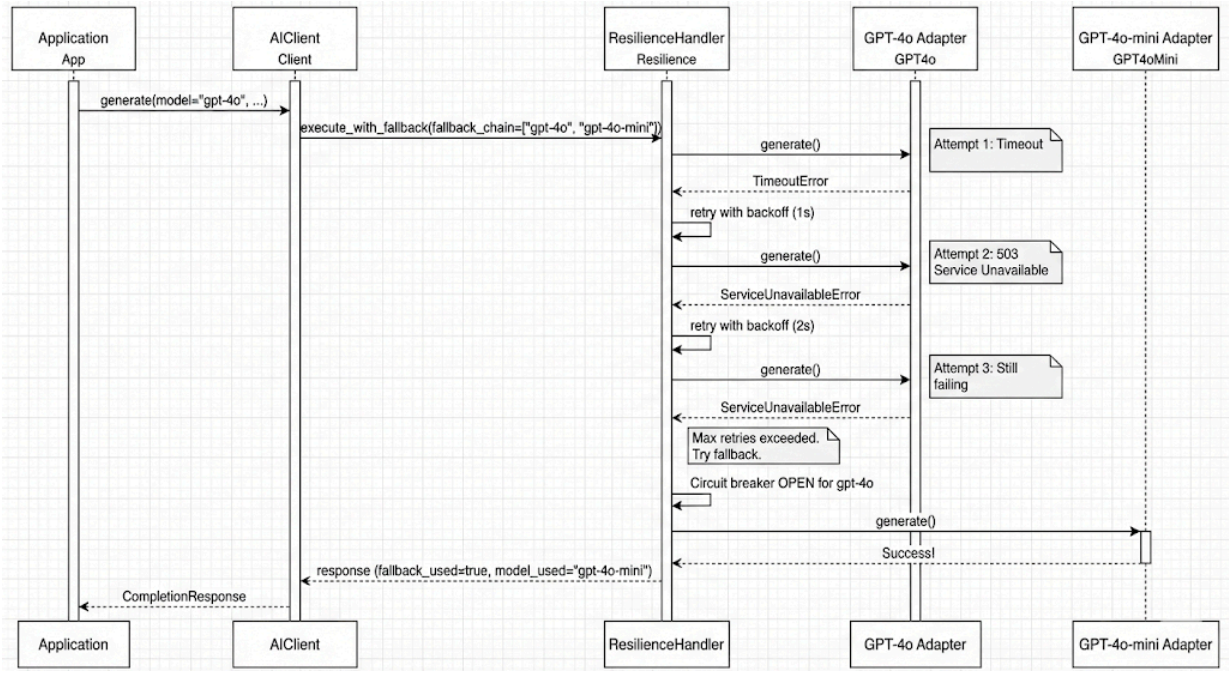
For real-time chat, we need streaming. Here's how it works:



Why this matters for EngageAI: Users see responses in real-time, word by word. No waiting 10 seconds for the full response.

4.3 Fallback Flow (When Things Go Wrong)

What happens when GPT-4o is down? We automatically try the next model:



Business value: Instead of showing users an error, we degrade gracefully. The extraction still works, just with a smaller model.

5 Why This Approach?

	Decision	Rationale
1	**Embedded library, not microservice**	Zero network latency. Critical for streaming and high-volume extraction.
2	**Azure Artifacts (private PyPI)**	Easy versioning, CI/CD integration, access control
3	**Semantic versioning**	Products can pin versions, upgrade when ready
4	**Backward compatibility**	We never break existing products

---

6. What Could Go Wrong (and How We'll Handle It)

6.1 Risk Assessment

	Risk	Likelihood	Impact	Mitigation
1	**Provider API breaks unexpectedly**	Medium	High	Adapter pattern isolates changes. Only one file to fix.
2	**Performance regression in extraction**	Low	High	Benchmark before/after. Keep existing code as fallback during Phase 1.

3	**Migration takes longer than planned**	Medium	Medium	Phased approach. Each phase delivers standalone value.
4	**New model doesn't work with our schema**	Medium	Medium	Feature detection in adapters. `supports_structured_output()` method.
5	**Rate limiting across products**	High	Medium	Global rate limiter. Quota management per client/product.
6	**Secret exposure in logs**	Low	Critical	Never log API keys. Mask in traces. Azure Key Vault with Managed Identity.

6.2 Rollback Strategy

Every phase has a clear rollback path:

- **Phase 1:** Keep old `llm_service.py` in place. Feature flag to switch between old and new.
- **Phase 2:** New providers are additive. GPT-4o still works. Can disable new adapters.
- **Phase 3:** Each product migrates independently. Can revert one without affecting others.

API Usage Examples

Basic text generation:

Python

```python
from inflow_ai_core import AIClient, CompletionRequest

client = AIClient()
response = await client.generate(CompletionRequest(
      model="gpt-4o",
      messages=[
          {"role": "system", "content": "You are a helpful assistant."},
          {"role": "user", "content": "Summarize this document..."}
      ],
      temperature=0.3,
      max_tokens=2000
))
print(response.content)

```

Structured output (extraction):

Python

```python
from inflow_ai_core import AIClient, StructuredRequest

schema = {
    "type": "object",
    "properties": {
        "invoice_number": {"type": "string"},
        "vendor": {"type": "string"},
        "total_amount": {"type": "number"}
    }
}

response = await client.generate_structured(StructuredRequest(
    model="gpt-4o",
    messages=[{"role": "user", "content": "Extract: ..."}],
```

```
    schema=schema
))
print(response.data)  # {"invoice_number": "INV-001", ...}

```

Streaming chat:

```python
async for chunk in client.stream(CompletionRequest(
    model="claude-sonnet-4.5",
    messages=[{"role": "user", "content": "Tell me a story..."}]
)):
    print(chunk.delta, end="", flush=True)

```

With fallback:

```python
response = await client.generate(
    CompletionRequest(model="gpt-5", messages=[...]),
    fallback_chain=["gpt-4o", "gpt-4o-mini"]
)
if response.fallback_used:
    print(f"Used fallback: {response.model_used}")

```

Technology Stack

	Component	Technology	Why
1	HTTP Client	`httpx` (async)	Modern async, connection pooling
2	OpenAI SDK	`openai>=1.0`	Official SDK with Azure support
3	Anthropic SDK	`anthropic>=0.20`	Official Claude SDK
4	Gemini SDK	`google-generativeai`	Official Google SDK
5	Validation	`pydantic>=2.0`	Type-safe models, JSON schema
6	Retry Logic	`tenacity`	Flexible retry/backoff
7	Tracing	`opentelemetry`	Distributed tracing
8	Logging	`structlog`	Structured logging