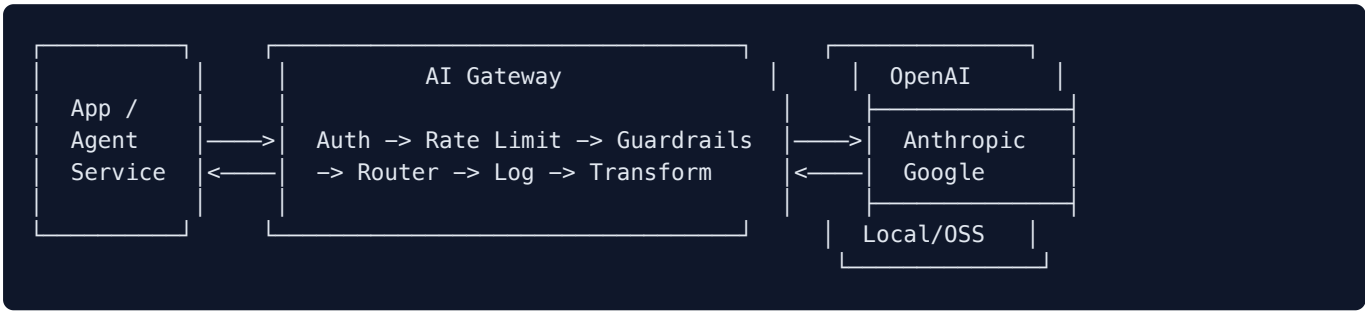
What Is an AI Gateway?

If your organization has more than one team calling LLM APIs, you need a gateway. An AI gateway sits between your application and LLM providers, centralizing cross-cutting concerns like authentication, rate limiting, routing, guardrails, and observability.



Core Components

Every AI gateway is built from the same building blocks. Understanding each component helps you decide whether to buy a managed gateway or build a custom one -- and what to prioritize first.

Component	Purpose	Implementation
Authentication	Validate API keys, JWT tokens	API key store, OAuth2
Rate limiting	Prevent abuse, manage quotas	Token bucket, sliding window
Input guardrails	Block harmful/invalid requests	PII scan, injection detection
Model router	Select provider/model per request	Rules, cost optimizer, fallback chain
Request transform	Normalize to provider format	OpenAI -> Anthropic format mapping
Response transform	Normalize provider responses	Unified response schema
Output guardrails	Filter harmful responses	Toxicity, PII, format validation
Logging/telemetry	Audit trail, analytics	Structured logs, traces
Caching	Reduce cost for repeated queries	Semantic cache, exact match cache
Cost tracking	Per-team/project attribution	Token counting, pricing tables

Gateway Products and Frameworks

The gateway market ranges from open-source proxies you run yourself to fully managed commercial platforms. Your choice depends on whether you need simple routing or enterprise features like audit logging and cost attribution.

Product	Type	Key Features
LiteLLM	Open source proxy	100+ providers, OpenAI-compatible API
Portkey	Commercial	Caching, fallbacks, load balancing
Helicone	Commercial	Logging, caching, rate limiting
Kong AI Gateway	Commercial	Enterprise, plugin-based
Cloudflare AI Gateway	Commercial	Edge caching, analytics
MLflow AI Gateway	Open source	Centralized credentials, rate limits
Semantic Router	Open source	Intent-based routing
Custom (FastAPI)	DIY	Full control

Authentication and Authorization

The gateway is the single point where you map internal team keys to provider credentials, enforce permissions, and set spending limits. Getting this right prevents unauthorized model access and surprise bills.

API Key Management

```
# Gateway authenticates app, not end-user
# Map internal keys to provider keys

KEY_MAP = {
    "app-key-team-alpha": {
        "team": "alpha",
        "budget_monthly_usd": 500,
        "allowed_models": ["gpt-4o", "claude-sonnet-4"],
        "rate_limit_rpm": 60
    }
}

def authenticate(request):
    api_key = request.headers.get("Authorization", "").replace("Bearer ", "")
    config = KEY_MAP.get(api_key)
    if not config:
        raise HTTPException(401, "Invalid API key")
    return config
```

Per-Team Permissions

Permission	Description	Example
`allowed_models`	Which models the team can use	`["gpt-4o-mini", "claude-haiku"]`
`max_tokens_per_request`	Cap output size	`4096`
`rate_limit_rpm`	Requests per minute	`100`

Permission	Description	Example
`rate_limit_tpm`	Tokens per minute	`100000`
`budget_monthly_usd`	Spending cap	`500`
`guardrail_level`	Strictness of content filtering	`"standard"` or `"strict"`

Rate Limiting

Without rate limiting, a single runaway script can burn through your entire monthly budget in hours. Rate limits protect against both abuse and accidental cost explosions.

Token Bucket Algorithm

```
import time

class TokenBucket:
    def __init__(self, capacity: int, refill_rate: float):
        self.capacity = capacity
        self.tokens = capacity
        self.refill_rate = refill_rate  # tokens per second
        self.last_refill = time.time()

    def consume(self, tokens: int = 1) -> bool:
        self._refill()
        if self.tokens >= tokens:
            self.tokens -= tokens
            return True
        return False

    def _refill(self):
        now = time.time()
        elapsed = now - self.last_refill
        self.tokens = min(self.capacity, self.tokens + elapsed * self.refill_rate)
        self.last_refill = now
```

Rate Limit Tiers

Tier	RPM	TPM	Max Concurrent	Use Case
Free	10	10K	2	Development
Standard	60	100K	10	Internal tools
Premium	300	500K	50	Production apps
Enterprise	1000+	2M+	200+	High-traffic services

Model Routing

Intelligent routing is where the gateway pays for itself -- sending simple queries to cheap models and complex ones to capable models can cut costs by 50-80% without sacrificing quality.

Routing Strategies

Strategy	Description	Best For
Cost-optimized	Route to cheapest model that can handle the task	Budget-conscious
Quality-first	Route to best model, fall back on failure	High-stakes tasks
Latency-first	Route to fastest responding model	Real-time apps
Load-balanced	Round-robin across providers	Even distribution
Capability-based	Route by task type (vision, code, etc.)	Multi-modal apps
Cascading	Try small model first, escalate if needed	Cost + quality balance

Router Implementation

```
ROUTING_TABLE = {
    "simple_qa": {
        "primary": "gpt-4o-mini",
        "fallback": ["claude-haiku-3.5", "gemini-flash"]
    },
    "complex_reasoning": {
        "primary": "claude-sonnet-4",
        "fallback": ["gpt-4o", "gemini-pro"]
    },
    "code_generation": {
        "primary": "claude-sonnet-4",
        "fallback": ["gpt-4o"]
    },
    "vision": {
        "primary": "gpt-4o",
        "fallback": ["gemini-pro", "claude-sonnet-4"]
    }
}

def classify_and_route(request):
    task_type = classify_task(request.messages)  # lightweight classifier
    config = ROUTING_TABLE[task_type]
    return config["primary"]
```

Cascading Pattern

```
async def cascade_call(request, models=["gpt-4o-mini", "gpt-4o", "claude-sonnet-4"]):
    """Try cheaper model first; escalate if response quality is low."""
    for model in models:
        response = await call_model(model, request)

        # Quality gate: check if response is good enough
        quality = assess_quality(request, response)
        if quality.score > 0.8:
```

```
            return response

        # If last model, return whatever we have
        if model == models[-1]:
            return response

    return response
```

Failover Patterns

LLM providers have outages -- and when your entire product depends on a single API, that outage becomes your outage. Multi-provider failover with circuit breakers keeps your application running through provider disruptions.

Pattern	Description	Recovery Time
Active-passive	Primary + standby	Seconds
Active-active	Multiple active providers	Instant (next request)
Circuit breaker	Disable failed provider temporarily	Configurable (30s-5m)
Retry with backoff	Retry on transient errors	Milliseconds-seconds
Hedge requests	Send to N providers, take first response	Instant

Circuit Breaker

```
class CircuitBreaker:
    def __init__(self, failure_threshold=5, recovery_timeout=60):
        self.failure_count = 0
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.last_failure = None
        self.state = "closed"  # closed=normal, open=failing, half_open=testing

    def call(self, fn, *args, **kwargs):
        if self.state == "open":
            if time.time() - self.last_failure > self.recovery_timeout:
                self.state = "half_open"
            else:
                raise CircuitOpenError()

        try:
            result = fn(*args, **kwargs)
            if self.state == "half_open":
                self.state = "closed"
                self.failure_count = 0
            return result
        except Exception as e:
            self.failure_count += 1
            self.last_failure = time.time()
            if self.failure_count >= self.failure_threshold:
```

```
                self.state = "open"
            raise
```

Caching

Caching is the easiest way to reduce LLM costs and latency simultaneously. For applications with repetitive queries, a semantic cache can eliminate 15-40% of LLM calls entirely.

Cache Type	Hit Rate	Latency Savings	Cost Savings
Exact match	Low (5-15%)	100% (no LLM call)	100%
Semantic cache	Medium (15-40%)	100% (no LLM call)	100%
KV cache (streaming)	N/A	30-50%	30-50%

Semantic Cache

```
def get_or_generate(query: str, threshold: float = 0.95):
    query_embedding = embed(query)

    # Search cache
    cached = cache_db.search(query_embedding, top_k=1)
    if cached and cached[0].score > threshold:
        return cached[0].response  # Cache hit

    # Cache miss: call LLM
    response = llm.generate(query)
    cache_db.insert(query_embedding, response, ttl=3600)
    return response
```

Cost Attribution

When multiple teams share LLM infrastructure, you need per-team cost tracking for budgeting, chargeback, and identifying optimization opportunities. Without attribution, nobody owns the bill.

Token-Based Cost Tracking

```
PRICING = {
    "gpt-4o":           {"input": 2.50, "output": 10.00},   # per 1M tokens
    "gpt-4o-mini":      {"input": 0.15, "output": 0.60},
    "claude-sonnet-4":  {"input": 3.00, "output": 15.00},
    "claude-haiku-3.5": {"input": 0.80, "output": 4.00},
}

def calculate_cost(model, input_tokens, output_tokens):
    prices = PRICING[model]
    cost = (input_tokens * prices["input"] + output_tokens * prices["output"]) / 1_000_000
    return round(cost, 6)
```

Cost Dashboard Metrics

Metric	Granularity	Purpose
Total spend	Daily/weekly/monthly	Budget tracking
Cost per team	Per team/project	Chargeback
Cost per request	Per request	Optimization
Model mix	% by model	Routing efficiency
Cache hit savings	Daily	ROI on caching
Cost per successful task	Per task type	Efficiency

Logging and Observability

You cannot debug, optimize, or audit what you do not log. Structured logging at the gateway layer gives you a single pane of glass across all LLM providers and teams.

What to Log

Field	Purpose
`request_id`	Correlation
`timestamp`	Timeline
`team_id`	Attribution
`model`	Routing analysis
`input_tokens`	Cost, monitoring
`output_tokens`	Cost, monitoring
`latency_ms`	Performance
`status_code`	Error tracking
`guardrail_triggered`	Safety monitoring
`cache_hit`	Efficiency
`cost_usd`	Finance

Do NOT log: Full prompt/response text in production (PII risk). Use a separate, access-controlled audit store if needed.

Common Pitfalls

Gateway mistakes are amplified across every request in your organization. A missing timeout or absent rate limit affects every team and every application simultaneously.

Pitfall	Problem	Fix
No failover	Single provider outage = total outage	Multi-provider with circuit breaker
Logging full prompts	PII exposure, storage cost	Log metadata only, separate audit store
No rate limiting	Cost explosion, abuse	Per-team token bucket
Static routing	Suboptimal cost/quality	Dynamic routing based on task
No cost alerts	Surprise bills	Budget caps with alerts at 50%, 80%, 100%
No caching	Paying for repeated queries	Semantic cache for common queries
Sync-only gateway	Bottleneck under load	Async processing, streaming passthrough
Missing timeout	Hung requests waste resources	30-120s timeouts on all provider calls