kAIxu Gateway — Implementation Directives
How To Wire Any App Through The Gate Instead of Calling AI Providers Directly

Gateway URL: `https://kaixu67.skyesoverlondon.workers.dev`
Auth: Bearer token in `Authorization` header
Format: JSON in, JSON out (or SSE for streaming)

Why Call The Gate Instead Of Gemini Directly

If you call Gemini directly from your app, the API key lives in your app — in environment variables, config files, or worse, bundled client-side code. Every developer who touches the project has access to it. Every leaked `.env` file costs you money.

The gateway owns the key. Your app gets a disposable token. If a token leaks, you revoke it without touching the Gemini key. If a developer leaves, you revoke their token. You can give different tokens to different apps or teams and track usage per token later.

Your app goes from this:
```
App → Google Gemini API (with your $$ key)
```
To this:
```
App → kAIxu Gate (with a cheap token) → Google Gemini API (key never leaves the server)
```

Same results. Zero key exposure.

The Four Endpoints

Method	Path	What it does
`GET`	`/v1/health`	Check the gateway is up and configured
`GET`	`/v1/models`	List available models
`POST`	`/v1/generate`	Full response — waits for complete answer, returns JSON
`POST`	`/v1/stream`	Streaming — returns SSE, bytes arrive in real time

Authentication

Every request needs an `Authorization` header:

```
Authorization: Bearer YOUR_TOKEN
```

Alternatively the header `X-KAIXU-TOKEN: YOUR_TOKEN` works too.

Where to get your token: ask whoever manages the gateway (`KAIXU_APP_TOKENS` env var). It's a comma-separated list — each value is a valid token. Tokens can be any string you want, e.g. `myapp-prod-v1`, a UUID, anything.

---

1. Health Check — Verify The Gate Is Reachable

Always a good first call to confirm your token is accepted.

curl
```bash
curl https://kaixu67.skyesoverlondon.workers.dev/v1/health \
  -H "Authorization: Bearer YOUR_TOKEN"
```

JavaScript (fetch)
```js
const res = await fetch('https://kaixu67.skyesoverlondon.workers.dev/v1/health', {
  headers: { 'Authorization': 'Bearer YOUR_TOKEN' }
});
const data = await res.json();
// { ok: true, keyConfigured: true, authConfigured: true, openGate: false }
```

Python (requests)
```python
import requests

r = requests.get(
    'https://kaixu67.skyesoverlondon.workers.dev/v1/health',
    headers={'Authorization': 'Bearer YOUR_TOKEN'}
)
print(r.json())
# {'ok': True, 'keyConfigured': True, 'authConfigured': True, 'openGate': False}
```

---

2. Basic Text Generation — `/v1/generate`

This is your main call for most use cases. Send a prompt, get back a complete answer.

The simplest possible call

curl
```bash
```

```
curl https://kaixu67.skyesoverlondon.workers.dev/v1/generate \
  -X POST \
  -H "Authorization: Bearer YOUR_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"input": {"type": "text", "content": "Explain WebSockets in one paragraph."}}'
```

JavaScript (fetch)
```js
const BASE = 'https://kaixu67.skyesoverlondon.workers.dev';
const TOKEN = 'YOUR_TOKEN';

async function ask(prompt) {
  const res = await fetch(`${BASE}/v1/generate`, {
    method: 'POST',
    headers: {
      'Authorization': `Bearer ${TOKEN}`,
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      input: { type: 'text', content: prompt }
    })
  });

  const data = await res.json();

  if (!data.ok) throw new Error(data.error);
  return data.text;
}

const answer = await ask('What is a REST API?');
console.log(answer);
```

Python
```python
import requests

BASE = 'https://kaixu67.skyesoverlondon.workers.dev'
TOKEN = 'YOUR_TOKEN'
HEADERS = {
    'Authorization': f'Bearer {TOKEN}',
    'Content-Type': 'application/json'
}

def ask(prompt: str) -> str:
    r = requests.post(
        f'{BASE}/v1/generate',
        headers=HEADERS,
        json={'input': {'type': 'text', 'content': prompt}}
```

```
    )
    data = r.json()
    if not data.get('ok'):
        raise Exception(data.get('error', 'Unknown error'))
    return data['text']

print(ask('Summarise the CAP theorem in plain English.'))
```

---

3. The Response Object

Every `/v1/generate` call returns this shape:

```json
{
  "ok": true,
  "model": "gemini-2.5-flash",
  "text": "The answer from the model...",
  "finishReason": "STOP",
  "usage": {
    "promptTokens": 14,
    "candidatesTokens": 120,
    "thoughtsTokens": 0,
    "totalTokens": 134
  }
}
```

Field	Type	Notes
`ok`	boolean	`false` if something went wrong — check `error` field
`model`	string	Which model actually answered
`text`	string	The answer — this is what you want
`finishReason`	string	`STOP` = complete answer. `MAX_TOKENS` = got cut off
`usage.promptTokens`	number	Tokens in your input
`usage.candidatesTokens`	number	Tokens in the answer
`usage.thoughtsTokens`	number	Internal reasoning tokens (gemini-2.5-pro only)
`usage.totalTokens`	number	Sum of all token usage

Error response:
```json
{
  "ok": false,
  "error": "Unauthorized"
}
```

---

4. Multi-Turn Conversations — Using `messages[]`

For chat interfaces, pass the full conversation history as `messages[]` — same format as OpenAI.

```js
const messages = [
  { role: 'user', content: 'My name is Skye.' },
  { role: 'assistant', content: 'Hello Skye, how can I help you?' },
  { role: 'user', content: 'What is my name?' }
];

const res = await fetch(`${BASE}/v1/generate`, {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${TOKEN}`,
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ messages })
});

const data = await res.json();
console.log(data.text); // "Your name is Skye."
```

The gateway converts `messages[]` to Gemini's `contents[]` format automatically. You don't need to think about Gemini's API format — send OpenAI-style `{ role, content }` objects and it works.

Building the messages array in a chat loop:
```js
const messages = [];

async function chat(userInput) {
  messages.push({ role: 'user', content: userInput });

  const res = await fetch(`${BASE}/v1/generate`, {
    method: 'POST',
    headers: { 'Authorization': `Bearer ${TOKEN}`, 'Content-Type': 'application/json' },
    body: JSON.stringify({ messages })
  });

  const data = await res.json();
  if (!data.ok) throw new Error(data.error);

  // Add the assistant reply to history so next turn has full context
  messages.push({ role: 'assistant', content: data.text });

  return data.text;
```

```
}

await chat('My favourite colour is indigo.');
await chat('What colour did I just mention?'); // "indigo"
```

---

5. System Instructions

Tell the model who it is and how to behave:

```js
const res = await fetch(`${BASE}/v1/generate`, {
  method: 'POST',
  headers: { 'Authorization': `Bearer ${TOKEN}`, 'Content-Type': 'application/json' },
  body: JSON.stringify({
    system: 'You are a concise legal assistant. Answer in plain English. Never give
    legal advice.',
    input: { type: 'text', content: 'What is habeas corpus?' }
  })
});
```

Per-request `system` overrides the gateway's global system prompt
(`KAIXU_GLOBAL_SYSTEM`). If you don't send one, the global default applies.

---

6. Choosing a Model

The gateway defaults to `gemini-2.5-flash`. To use a different model, pass `model`:

```js
body: JSON.stringify({
  model: 'gemini-2.5-pro',   // the thinking model — smarter, slower
  input: { type: 'text', content: 'Analyse the tradeoffs of microservices vs monoliths.'
  }
})
```

When to use which:

Use Case	Model
Most things	`gemini-2.5-flash` (default)
Complex reasoning, code review, long analysis	`gemini-2.5-pro`

Get the model list dynamically:
```js
```

```
const res = await fetch(`${BASE}/v1/models`, {
  headers: { 'Authorization': `Bearer ${TOKEN}` }
});
const { models } = await res.json();
// models = [{ id: 'gemini-2.5-flash', ... }, { id: 'gemini-2.5-pro', ... }]
```

---

7. Generation Config — Temperature, Token Limits, etc.

Pass `generationConfig` and it goes straight to Gemini verbatim:

```js
body: JSON.stringify({
  input: { type: 'text', content: 'Write me a short poem.' },
  generationConfig: {
    temperature: 1.2,          // 0.0 = deterministic, 2.0 = wildly creative
    topP: 0.95,
    maxOutputTokens: 512,      // cap the answer length
    stopSequences: ['END']     // stop generating if this string appears
  }
})
```

No server-side caps are applied — whatever you send in `generationConfig` is what Gemini gets.

Note on `gemini-2.5-pro`: This model thinks internally before answering. If you set a low `maxOutputTokens`, it can burn all the tokens on thinking and return nothing. For `gemini-2.5-pro`, either omit `maxOutputTokens` entirely or set it high (`32768` or more).

---

8. JSON Output Mode

Ask the model to return structured JSON instead of free text:

```js
const res = await fetch(`${BASE}/v1/generate`, {
  method: 'POST',
  headers: { 'Authorization': `Bearer ${TOKEN}`, 'Content-Type': 'application/json' },
  body: JSON.stringify({
    output: { format: 'json' },
    input: {
      type: 'text',
      content: 'Return a JSON object with fields: name, capital, population for France.'
    }
  })
```

```
});

const data = await res.json();
const country = JSON.parse(data.text);
// { name: 'France', capital: 'Paris', population: 68000000 }
```

---

9. Streaming — `/v1/stream`

Use streaming when:
- Answers take more than 1-2 seconds
- You want the UI to feel responsive as words appear
- You're using `gemini-2.5-pro` (which can take 10-30s before first byte)

The endpoint returns Server-Sent Events (SSE). Each `data:` line is a raw Gemini chunk.

JavaScript — ReadableStream
```js
async function streamAnswer(prompt, onChunk) {
  const res = await fetch(`${BASE}/v1/stream`, {
    method: 'POST',
    headers: {
      'Authorization': `Bearer ${TOKEN}`,
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      input: { type: 'text', content: prompt }
    })
  });

  if (!res.ok) throw new Error(`HTTP ${res.status}`);

  const reader = res.body.getReader();
  const decoder = new TextDecoder();
  let buffer = '';

  while (true) {
    const { done, value } = await reader.read();
    if (done) break;

    buffer += decoder.decode(value, { stream: true });
    const lines = buffer.split('\n');
    buffer = lines.pop(); // keep incomplete line for next chunk

    for (const line of lines) {
      if (!line.startsWith('data: ')) continue;
      const raw = line.slice(6).trim();
      if (raw === '[DONE]') return;
```

```
      try {
        const chunk = JSON.parse(raw);
        const text = chunk.candidates?.[0]?.content?.parts
          ?.filter(p => !p.thought)        // skip internal thinking
          ?.map(p => p.text || '')
          ?.join('') || '';
        if (text) onChunk(text);
      } catch {}
    }
  }
}

// Usage: stream tokens into a <div> as they arrive
let output = '';
await streamAnswer('Write a 500-word essay on stoicism.', chunk => {
  output += chunk;
  document.getElementById('output').textContent = output;
});
```

Python — httpx streaming
```python
import httpx
import json

BASE = 'https://kaixu67.skyesoverlondon.workers.dev'
TOKEN = 'YOUR_TOKEN'

def stream_answer(prompt: str):
    with httpx.stream(
        'POST',
        f'{BASE}/v1/stream',
        headers={
            'Authorization': f'Bearer {TOKEN}',
            'Content-Type': 'application/json'
        },
        json={'input': {'type': 'text', 'content': prompt}},
        timeout=None   # no timeout — let the stream complete
    ) as r:
        buffer = ''
        for chunk in r.iter_text():
            buffer += chunk
            while '\n' in buffer:
                line, buffer = buffer.split('\n', 1)
                if not line.startswith('data: '):
                    continue
                raw = line[6:].strip()
                if raw == '[DONE]':
                    return
```

```
                try:
                    data = json.loads(raw)
                    parts = data.get('candidates', [{}])[0].get('content',
                    {}).get('parts', [])
                    text = ''.join(p.get('text', '') for p in parts if not
                    p.get('thought'))
                    if text:
                        print(text, end='', flush=True)
                except Exception:
                    pass

stream_answer('Explain the theory of general relativity step by step.')
```

---

10. Migrating From Direct Gemini API Calls

If your code currently calls Gemini directly, here's the mapping:

Before (direct Gemini):
```js
const res = await fetch(
  `https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-flash:generateCont
  ent?key=${GEMINI_KEY}`,
  {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      contents: [{ role: 'user', parts: [{ text: 'Hello' }] }]
    })
  }
);
const raw = await res.json();
const text = raw.candidates[0].content.parts[0].text;
```

After (through the gate):
```js
const res = await fetch(`${BASE}/v1/generate`, {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${TOKEN}`,  // ← replaces API key
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    input: { type: 'text', content: 'Hello' }  // ← simplified format
  })
});
const data = await res.json();
```

```
const text = data.text;  // ← already extracted for you
```

Or if you want to pass Gemini's native `contents[]` format directly, you can do that too — the gateway passes it through unchanged:
```js
body: JSON.stringify({
  contents: [{ role: 'user', parts: [{ text: 'Hello' }] }]
})
```

---

11. Migrating From OpenAI Calls

The gateway uses OpenAI-style `messages[]` format, so most of your structure stays the same — just change the URL and auth:

Before (OpenAI):
```js
const res = await fetch('https://api.openai.com/v1/chat/completions', {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${OPENAI_KEY}`,
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    model: 'gpt-4o',
    messages: [
      { role: 'system', content: 'You are a helpful assistant.' },
      { role: 'user', content: 'Hello' }
    ]
  })
});
const data = await res.json();
const text = data.choices[0].message.content;
```

After (kAIxu gate):
```js
const res = await fetch(`${BASE}/v1/generate`, {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${TOKEN}`,  // ← different token, same header
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    // system message goes in the 'system' field, not in messages[]
    system: 'You are a helpful assistant.',
    messages: [
```

```
      { role: 'user', content: 'Hello' }
    ]
  })
});
const data = await res.json();
const text = data.text;  // ← data.text instead of data.choices[0].message.content
```

What changes:
- URL: `api.openai.com` → `kaixu67.skyesoverlondon.workers.dev`
- Token: OpenAI key → your gate token (in the same `Authorization: Bearer` header)
- `model`: `gpt-4o` → `gemini-2.5-flash` or `gemini-2.5-pro`
- System message: move from `messages[]` to the top-level `system` field
- Response: `data.choices[0].message.content` → `data.text`

---

12. Environment Setup For Teams

Never hardcode the token. Store it as an environment variable and load it at runtime.

Node.js / Next.js / Vite
```js
// .env
VITE_KAIXU_TOKEN=your_token_here
VITE_KAIXU_BASE=https://kaixu67.skyesoverlondon.workers.dev

// kaixu.js — re-usable client
const BASE  = import.meta.env.VITE_KAIXU_BASE;
const TOKEN = import.meta.env.VITE_KAIXU_TOKEN;

export async function ask(prompt, options = {}) {
  const res = await fetch(`${BASE}/v1/generate`, {
    method: 'POST',
    headers: { 'Authorization': `Bearer ${TOKEN}`, 'Content-Type': 'application/json' },
    body: JSON.stringify({ input: { type: 'text', content: prompt }, ...options })
  });
  const data = await res.json();
  if (!data.ok) throw new Error(data.error);
  return data;
}
```

Python / Django / FastAPI
```python
# .env
KAIXU_TOKEN=your_token_here
KAIXU_BASE=https://kaixu67.skyesoverlondon.workers.dev

# kaixu.py — re-usable client
```

```
import os, requests

BASE  = os.environ['KAIXU_BASE']
TOKEN = os.environ['KAIXU_TOKEN']
HEADERS = {'Authorization': f'Bearer {TOKEN}', 'Content-Type': 'application/json'}

def ask(prompt: str, **kwargs) -> dict:
    r = requests.post(
        f'{BASE}/v1/generate',
        headers=HEADERS,
        json={'input': {'type': 'text', 'content': prompt}, **kwargs}
    )
    data = r.json()
    if not data.get('ok'):
        raise Exception(data.get('error'))
    return data
```

React Native / Expo
```js
// app.config.js / eas.json secrets or expo-constants
import Constants from 'expo-constants';
const BASE  = Constants.expoConfig.extra.kaixuBase;
const TOKEN = Constants.expoConfig.extra.kaixuToken;
```

Never commit tokens. Add `.env` to `.gitignore`. On CI/CD (Vercel, Netlify, etc.), add the token as an environment variable in the platform dashboard.

---

13. Error Handling — What The Gate Returns

HTTP Status	`ok`	When
200	`true`	Success
400	`false`	Bad request body — missing required fields
401	`false`	Missing or invalid token
405	`false`	Wrong HTTP method
413	`false`	Request body too large (>5MB default)
500	`false`	Gateway misconfigured or Gemini error

Robust error handling:
```js
async function safeAsk(prompt) {
  try {
    const res = await fetch(`${BASE}/v1/generate`, {
      method: 'POST',
      headers: { 'Authorization': `Bearer ${TOKEN}`, 'Content-Type': 'application/json'
      },
```

```
      body: JSON.stringify({ input: { type: 'text', content: prompt } })
    });

    const data = await res.json();

    if (!res.ok || !data.ok) {
      console.error('Gate error:', data.error);
      return null;
    }

    if (data.finishReason === 'MAX_TOKENS') {
      console.warn('Response was cut off — try raising maxOutputTokens in
      generationConfig');
    }

    return data.text;
  } catch (err) {
    console.error('Network error:', err.message);
    return null;
  }
}
```

---

14. Full Request Body Reference

All fields are optional except one of `input`, `messages`, or `contents`:

```js
{
  // --- Model ---
  "model": "gemini-2.5-flash",          // default: KAIXU_DEFAULT_MODEL

  // --- Input (pick one) ---
  "input": { "type": "text", "content": "Your prompt" },   // simple text
  "messages": [{ "role": "user", "content": "..." }],      // OpenAI format
  "contents": [{ "role": "user", "parts": [...] }],        // raw Gemini format

  // --- Behaviour ---
  "system": "You are a helpful assistant.",    // override system instruction
  "generationConfig": {                        // passed to Gemini verbatim
    "temperature": 0.7,
    "maxOutputTokens": 8192,
    "topP": 0.9,
    "stopSequences": []
  },
  "output": { "format": "json" },            // json | text | markdown

  // --- Advanced ---
```

```
  "safetySettings": [...],                    // Gemini safety config
  "tools": [...],                             // function calling tools
  "toolConfig": {...},                        // tool config
  "includeRaw": false                         // true = include raw Gemini response at
  .raw
}
```

Quick Start Checklist

- [ ] Get your bearer token from the gateway owner
- [ ] Store it in an environment variable — never in code
- [ ] Hit `/v1/health` first to confirm auth works
- [ ] Use `input.content` for simple prompts
- [ ] Use `messages[]` for multi-turn chat (OpenAI format)
- [ ] Use `model: "gemini-2.5-pro"` only when you need deep reasoning
- [ ] Use `/v1/stream` for anything that might take more than 2 seconds
- [ ] Check `data.ok` and `data.finishReason` in every response
- [ ] Never hardcode the token — environment variables only

---

kAIxu Gate Delta — Skyes Over London LC — 2026