Phase 4 Training: Workers AI + Vectorize + AI Gateway

What We Built

A single feature: **"Ask This Blog"** — a conversational AI interface that lets visitors ask questions about saltwaterbrc.com's blog content and get AI-generated answers in seconds.

Try it now: **saltwaterbrc.com/ask.html**

Type a question like "What is a Durable Object?" or "Why are you building on Cloudflare?" and the system searches the blog, finds relevant content, and generates an answer using AI.

Component	Product	What It Does
Content indexing	Workers AI + Vectorize	Turns blog posts into mathematical vectors (embeddings) for semantic search
Semantic search	Vectorize	Finds the 3-5 most relevant blog chunks based on meaning (not keywords)
Answer generation	Workers AI	Uses Llama 3.1 8B to synthesize an answer from the relevant chunks
Observability	AI Gateway	Logs every AI request, tracks token usage, enables caching and rate limiting

Live Demo

Behind the scenes:

- 14 blog content chunks stored in Vectorize
- Each chunk is a 768-dimensional vector (the "meaning" of that text)
- User's question gets embedded the same way
- Vectorize finds the closest vectors (most semantically similar content)
- Llama 3.1 8B generates an answer from that context
- Every API call routed through AI Gateway for logging and caching

The Three Products Explained

1. Workers AI: Edge Inference for Large Language Models

What It Is

Compare to OpenAI's API:

- OpenAI: Your request goes to OpenAI's servers in the cloud → response comes back
- Workers AI: Your request hits Cloudflare's nearest edge server → inference runs right there → response in milliseconds

The Two Models We Use

1. **bge-base-en-v1.5** (text embedding model) - Takes any text as input - Outputs a 768-dimensional vector (array of 768 numbers representing the meaning of that text) - Used to index blog content and embed user questions - Think of it as "fingerprinting" text for semantic similarity

2. **Llama 3.1 8B Instruct** (text generation model) - Takes a prompt (usually: "Here is context about [topic]. Answer this question: [question]") - Outputs natural language text - The Instruct variant is fine-tuned for following instructions and answering questions - Same model used by companies like Google and Meta in production

How You Call It in Code

One line of code:

```
// Embed text const embedding = await env.AI.run("@cf/baai/bge-base-en-v1.5", { text: "What is a Durable Object?" }); // Returns: { data: [[0.123, -0.456, ..., 0.789]] } // Generate answer const response = await env.AI.run("@cf/meta/llama-3.1-8b-instruct", { messages: [ { role: "system", content: "You are a helpful assistant." }, { role: "user", content: "Answer this: [question]" } ] }); // Returns: { result: [{ text: "A Durable Object is..." }] }
```

The Big Difference from External APIs

- **Your data stays on Cloudflare's network.** No request to OpenAI, no logs stored elsewhere, no third-party access.
- **No rate limiting from OpenAI.** Cloudflare's rate limits apply, not theirs.
- **Predictable pricing.** Pay per token — no monthly subscription, no overage surprises.
- **Inference at the edge.** Sub-100ms latency because models run in your regional data centers.

Customer Pitch

"You're building AI features but worried about data privacy, vendor lock-in, or API costs. Workers AI runs on Cloudflare's network, your data never leaves. Same inference speed, lower cost, and you control the infrastructure."

2. Vectorize: Vector Database for Semantic Search

What It Is

A **managed vector database** — stores content as mathematical vectors and finds similar content by meaning (not keyword matching).

Here's the problem Vectorize solves:

- Search for "durable" → only returns pages with the word "durable"
- Miss pages about "persistent state" or "consistent storage" that mean the same thing
- Search for "how do I store state across requests?" → understands the intent
- Returns pages about Durable Objects, Workers KV, D1, databases — all relevant even if they don't use those exact keywords

How It Works

1. **Index content**: Take blog chunks (articles, sections), embed them with bge-base-en-v1.5, store the vectors in Vectorize 2. **User asks a question**: Embed the question the same way → get a vector 3. **Search**: Vectorize finds the closest vectors (highest cosine similarity) → returns those blog chunks 4. **Context window**: Pass the top 5 chunks to Llama 3.1 8B as context → it generates an answer

Creating an Index

```
npx wrangler vectorize create saltwaterbrc-blog --preset @cf/baai/bge-base-en-v1.5
```

This creates a vector index optimized for the bge-base-en-v1.5 model (768-dimensional). Visible in the Cloudflare dashboard — you can see index size, manage it, clear it.

Querying from a Worker

```
// Get the embedding for the user's question const questionEmbedding = await env.AI.run("@cf/baai/bge-base-en-v1.5", { text: userQuestion }); // Search the index (top 5 most relevant chunks) const results = await env.VECTORIZE_INDEX.query( questionEmbedding.data[0], { topK: 5, returnMetadata: "all" } ); // results[0].metadata.text = the most relevant blog chunk // results[1].metadata.text = second most relevant // etc.
```

Why This Is Powerful

- **No full-text search needed.** Keyword search is fast but dumb. Semantic search understands meaning.
- **Scales to any knowledge base.** Product docs, internal wikis, customer support tickets, medical records — if it can be chunked and embedded, Vectorize can index it.
- **Enables RAG (Retrieval Augmented Generation).** This is the foundation of "ask your docs" features that enterprises are building.

Enterprise Use Cases

- **E-commerce product discovery**: Embed product descriptions → customers search by intent ("waterproof hiking boots") instead of exact keywords
- **Internal knowledge base**: Index employee handbook, code docs, procedures → anyone can ask questions in natural language
- **Support chatbots**: Index all support articles → chat can answer common questions instantly
- **Medical records**: Index patient histories (HIPAA-compliant) → doctors search semantically

Customer Pitch

"Vector databases are the retrieval layer for AI apps. Product discovery, content recommendations, support chatbots — all powered by semantic search. Vectorize handles it at the edge, no external database needed."

3. AI Gateway: Observability and Control Plane

What It Is

A centralized **logging, caching, and rate-limiting layer for AI API calls**. Set up in 30 seconds. Gives you full visibility into every request your app makes to Workers AI.

Think of it like Cloudflare's Web Analytics but for AI — instead of HTTP requests, you're logging AI inferences.

What It Gives You (Out of the Box)

1. **Request logging**: Every AI call logged with timestamp, model, tokens used, latency, cost 2. **Analytics dashboard**: Charts showing requests over time, token usage trends, cost breakdowns 3. **Caching**: Identical prompts served from cache (zero inference cost if cached) 4. **Rate limiting**: Protect against abuse or runaway requests 5. **Data Loss Prevention (DLP)**: Scan prompts and responses for sensitive data (credit cards, SSNs, PII) 6. **Cost tracking**: See exactly what each gateway costs, per request

Setup (30 seconds)

```
// Before (no logging): const response = await env.AI.run("@cf/meta/llama-3.1-8b-instruct", { messages: [...] }); // After (with logging through AI Gateway): const response = await env.AI.run("@cf/meta/llama-3.1-8b-instruct", { messages: [...] }, { gateway: { id: "saltwaterbrc" } });
```

That's it. Every request now flows through AI Gateway.

The Dashboard

Go to **Workers** → **AI Gateway** → **saltwaterbrc** and see:

- **Requests**: 1,234 total, 156 today, 12 in last hour
- **Tokens**: 45,678 prompt tokens, 23,456 completion tokens
- **Cost**: $2.34 this month
- **Cache hit rate**: 34% (meaning 34% of requests served from cache, zero cost)
- **Model breakdown**: Which models are being called, how many times each

Caching Example

Let's say a user asks "What is a Durable Object?" and the Worker logs this AI request through AI Gateway:

```
Request 1: "What is a Durable Object?" → AI Gateway receives the request → Checks cache — miss → Calls Llama 3.1 8B → Gets answer back → Caches it → Returns to user → Cost: $0.0005 (or whatever the token count is) Request 2 (same question from a different user, 30 seconds later): → AI Gateway receives the request → Checks cache — HIT → Returns cached answer instantly → Cost: $0.00 (cached, zero tokens consumed)
```

That's a 50% cost reduction if 50% of questions are duplicates (which in real apps, many are).

Customer Pitch

"You wouldn't run a database without monitoring. Why would you run AI without it? AI Gateway gives you request logging, caching, rate limiting, and cost tracking — no extra code needed."

The Code: Building "Ask This Blog"

File Structure

```
SaltWaterBRC/ ■■■ ask-worker/ # Standalone Worker (separate project) ■ ■■■ src/ ■ ■ ■■■ index.js # Embedding, search, generation logic ■ ■■■ wrangler.toml # Bindings: AI, Vectorize, AI Gateway ■ ■■■ package.json ■■■ site/ ■ ■■■ ask.html # Frontend UI (simple HTML form) ■ ■■■ ...
```

The **ask-worker** is a standalone Worker, not a Pages Function. Same reason as Phase 3's counter-worker: complex bindings (AI + Vectorize) work more reliably as standalone Workers called directly from the frontend.

wrangler.toml

```
name = "saltwaterbrc-ask" main = "src/index.js" compatibility_date = "2025-02-16" [ai] binding = "AI" [[vectorize]] binding = "VECTORIZE_INDEX" index_name = "saltwaterbrc-blog"
```

Two bindings:

- `AI` — gives you `env.AI.run()` for embeddings and generation
- `VECTORIZE_INDEX` — gives you `env.VECTORIZE_INDEX.query()` for semantic search

The Three Key Operations

1. Embed Content (Indexing)

Run once when you first set up the system. Takes blog posts, chunks them, embeds each chunk, stores in Vectorize.

```
const chunks = [ "A Durable Object is a small piece of code that holds persistent state...", "Each Durable Object instance runs in a single location for consistency...", // ... 12 more chunks ]; for (const chunk of chunks) { // Embed the chunk const embedding = await env.AI.run("@cf/baai/bge-base-en-v1.5", { text: chunk }, { gateway: { id: "saltwaterbrc" } }); // Insert into Vectorize with metadata (the original text) await env.VECTORIZE_INDEX.insert([{ id: `chunk-${i}`, values: embedding.data[0], metadata: { text: chunk } }]); }
```

This is a one-time operation. 14 blog chunks, 14 API calls to embed them. Cost: ~$0.01.

2. Search with Semantic Similarity

User asks a question. Embed it. Find the top 5 most similar chunks. This is instant and free (uses the vector database, not an AI model).

```
// User's question comes in const question = "What is a Durable Object?"; // Embed the question const questionEmbedding = await env.AI.run("@cf/baai/bge-base-en-v1.5", { text: question }, { gateway: { id: "saltwaterbrc" } }); // Search Vectorize (instant, no model inference) const results = await env.VECTORIZE_INDEX.query( questionEmbedding.data[0], { topK: 5, returnMetadata: "all" } ); // results = [ // { id: "chunk-0", score: 0.95, metadata: { text: "A Durable Object is..." } }, // { id: "chunk-3", score: 0.87, metadata: { text: "Each DO instance..." } }, // ... // ]
```

The score is cosine similarity (0-1). Higher = more semantically similar.

3. Generate Answer with Context

Take the top 5 chunks and feed them to Llama 3.1 8B as context. The model synthesizes an answer based on the context, not from its training data alone (this is the "Retrieval Augmented Generation" part — RAG).

```
// Build the context from the search results const context = results .map(r => r.metadata.text) .join("\n\n"); // Ask Llama to answer using the context const answer = await env.AI.run("@cf/meta/llama-3.1-8b-instruct", { messages: [ { role: "system", content: "You are a helpful assistant. Answer the question based on the provided context. If the context doesn't contain the answer, say so." }, { role: "user", content: `Context:\n${context}\n\nQuestion: ${question}` } ] }, { gateway: { id: "saltwaterbrc" } }); // answer.result[0].text = the generated answer return new Response(JSON.stringify({ answer: answer.result[0].text, sources: results.map(r => r.id) }), { headers: { "Content-Type": "application/json" } });
```

Full Request Flow

1. User submits question on ask.html 2. Frontend calls `/ask?q=what+is+a+durable+object` 3. Worker receives request 4. Embed question (Workers AI + AI Gateway) 5. Search Vectorize (instant semantic search) 6. Generate answer (Workers AI + Llama + AI Gateway) 7. Return JSON with answer and source chunks 8. Frontend displays answer to user

Total latency: ~500ms (embedding + search + generation). Total cost: ~$0.0003 per question (token usage for 2 AI calls). Cached queries: ~50ms, $0 cost.

Deployment Steps

Step 1: Create the Vectorize Index

```
npx wrangler vectorize create saltwaterbrc-blog --preset @cf/baai/bge-base-en-v1.5
```

Output:

```
✓ Created vector index saltwaterbrc-blog ID: a1b2c3d4-e5f6-7890-abcd-ef1234567890 Dimension: 768 Metric: cosine
```

Step 2: Create AI Gateway in Dashboard

1. Go to **Cloudflare Dashboard** → **Workers** → **AI Gateway** 2. Click **Create Gateway** 3. Name: `saltwaterbrc` 4. Click **Create**

Step 3: Deploy the Worker

```
cd ~/Documents/Claude/SaltWaterBRC/ask-worker npm install npx wrangler deploy
```

Output:

```
✓ Published saltwaterbrc-ask URL: https://saltwaterbrc-ask.saltwaterbrc.workers.dev
```

Step 4: Ingest Blog Content

Call the `/ingest` endpoint to chunk, embed, and store all blog content:

```
curl https://saltwaterbrc-ask.saltwaterbrc.workers.dev/ingest
```

This:

- Reads blog posts from the Pages site
- Chunks them into semantic pieces (paragraphs, sections)
- Embeds each chunk
- Stores in Vectorize

Time: ~30 seconds Cost: ~$0.02 (14 chunks × embedding cost)

Step 5: Test the API

```
curl "https://saltwaterbrc-ask.saltwaterbrc.workers.dev/ask?q=what%20is%20a%20durable%20object"
```

Response:

```
{ "answer": "A Durable Object is a small, consistent piece of code that holds persistent state...", "sources": ["chunk-0", "chunk-3", "chunk-7"], "latency_ms": 487 }
```

Step 6: Deploy Frontend (ask.html)

The ask.html page already calls the Worker endpoint. Push to GitHub:

```
cd ~/Documents/Claude/SaltWaterBRC/site git add ask.html git commit -m "Add Ask This Blog feature" git push # Auto-deploys via Cloudflare Pages
```

Why a Standalone Worker (Not Pages Function)

Same lesson from Phase 3: **Pages Functions with complex bindings cause deployment issues.**

The ask-worker has three bindings (AI, Vectorize, AI Gateway) that interact with Models API, vector database, and observability service. When we tried bundling this into Pages Functions, we hit:

- "Failed to resolve binding" errors
- Vectorize binding not getting picked up
- AI Gateway configuration conflicts

ask.html snippet:

```
fetch(`https://saltwaterbrc-ask.saltwaterbrc.workers.dev/ask?q=${encodeURIComponent(question)}`) .then(r => r.json()) .then(data => { document.getElementById("answer").innerText = data.answer; });
```

Key Concepts

Embeddings: How Meaning Becomes Math

An embedding is a vector — an array of numbers that represents the meaning of text.

The bge-base-en-v1.5 model turns text into a 768-dimensional vector:

```
Text: "A Durable Object holds persistent state" ↓ Embedding: [0.123, -0.456, 0.789, ..., 0.234] ← 768 numbers Text: "Persistent state at the edge" ↓ Embedding: [0.125, -0.450, 0.792, ..., 0.231] ← slightly different (similar meaning)
```

These two embeddings are "close" in vector space (high cosine similarity) because the texts mean similar things.

This is why semantic search works: you embed the question and the documents, then find which document embeddings are closest to the question embedding.

Chunking: Breaking Content Into Pieces

You can't embed an entire blog post (it'd be too long, too much meaning mixed). Instead, chunk it:

- Full article: "The Roadmap: What I'm Building on Cloudflare"
- Chunks:

1. "Introduction: Why Cloudflare? Three reasons..." 2. "Phase 1: Workers and Pages..." 3. "Phase 2: Durable Objects and R2..." 4. "Phase 3: The Learning Curve..." 5. "Phase 4: RAG and Workers AI..."

Each chunk gets embedded separately. When a user asks, semantic search finds relevant chunks (not the whole article).

Cosine Similarity: Measuring "Closeness"

Cosine similarity measures how aligned two vectors are (0 = opposite, 1 = identical).

If user asks "How do I store state?" and a chunk talks about "persistent state storage," they'd score ~0.92. If another chunk talks about "streaming video quality," it'd score ~0.15.

Vectorize returns results sorted by cosine similarity, highest first. The top 5 are almost always relevant.

Context Window: The RAG Pattern

Large language models have a "context window" — the maximum amount of text they can read and reason about at once.

Llama 3.1 8B has an 8k token context window (roughly 6,000 words). You don't feed it your entire knowledge base. You feed it:

1. System prompt (instructions) 2. User question (100 tokens) 3. Top 5 relevant chunks (2,000 tokens) 4. Blank space for the model to write the answer (remaining tokens)

This is **Retrieval Augmented Generation**. The model is "augmented" by retrieval — it doesn't answer from training data alone, it answers from the context you provide.

Benefits:

- Answers are based on your data (more accurate)
- No hallucination (the model can't make up information not in the context)
- You control what the model knows (add new blog post → ingest it → model knows about it instantly)

Customer Pitch Angles

Angle 1: "Ask Your Docs"

"Your knowledge base is locked inside static documentation. Your customers dig through 50 pages to find one answer. What if they could just ask? 'How do I authenticate an API call?' and get an instant answer?"

Angle 2: Data Privacy and Compliance

"You want AI features but can't use ChatGPT — data privacy concerns, compliance requirements, or you just don't trust third parties. Workers AI runs on Cloudflare's network. Your data never leaves. Your docs stay internal."

Angle 3: Cost and Speed

"Building with OpenAI's API means paying per request and dealing with latency from cloud-to-cloud. Workers AI runs at the edge, in the data center serving your users. Faster and cheaper."

- OpenAI GPT-4: $0.03 per 1k input tokens
- Llama 3.1 8B on Workers AI: $0.0001 per 1k input tokens
- 300x cost difference. Plus 300ms faster (edge inference vs. cloud round-trip).

Angle 4: Built-In Observability

"ChatGPT costs you money every time someone asks a question. You don't know if people are using it, what they're asking, or how much it costs. AI Gateway logs everything: requests, tokens, cost, cache hits."

What's Different from Using OpenAI

Aspect	OpenAI API	Workers AI
Inference location	OpenAI's cloud (virginia, maybe)	Cloudflare's edge (300+ cities)
Data residency	Leaves your network → OpenAI servers	Stays on Cloudflare's network
Latency	200-500ms (cloud round-trip)	50-150ms (edge inference)
Cost model	Pay per token (recurring)	Pay per token + free tier
Observability	Use Helicone, Langsmith, etc.	AI Gateway (built-in)
Caching	Need external caching layer	AI Gateway caches automatically
Rate limiting	Need external rate limiter	AI Gateway rate limiting built-in
Model selection	OpenAI models (GPT-4, etc.)	Open models (Llama, Mistral, etc.)
Compliance	Varies (ChatGPT not HIPAA)	HIPAA-eligible infrastructure

Deployment Gotchas (Lessons from Real Errors)

1. Vectorize Index Must Exist Before Worker Deploys

If you deploy the ask-worker before creating the Vectorize index, binding resolution fails.

2. AI Gateway Setup is Separate from Worker Deployment

Creating a gateway in the dashboard doesn't automatically connect it to your Worker. You need the code change:

```
// Without this, requests go to AI but NOT through AI Gateway env.AI.run("@cf/baai/bge-base-en-v1.5", { text: ... });
// With this, requests route through AI Gateway (logging, caching, rate limiting)
env.AI.run("@cf/baai/bge-base-en-v1.5", { text: ... }, { gateway: { id: "saltwaterbrc" } });
```

If you add the `gateway` option but the gateway doesn't exist in the dashboard, you'll get an error. Always create the gateway first.

3. Ingest Endpoint Needs Authentication

The `/ingest` endpoint is dangerous — it's expensive to run (embeds all blog content). If it's public, anyone can spam it.

```
// Bad (anyone can call it) export default { async fetch(request) { if (request.url.includes("/ingest")) { return
ingestBlogContent(env); } } }; // Good (requires a secret token) export default { async fetch(request) { if
(request.url.includes("/ingest")) { const token = request.headers.get("Authorization"); if (token !== `Bearer
${env.INGEST_TOKEN}`) { return new Response("Unauthorized", { status: 401 }); } return ingestBlogContent(env); } } };
```

Set `INGEST_TOKEN` in Cloudflare dashboard → Workers → Settings → Environment Variables.

Then call it:

```
curl https://saltwaterbrc-ask.saltwaterbrc.workers.dev/ingest \ -H "Authorization: Bearer your-secret-token"
```

4. Vectorize Queries Can Return No Results

If the user asks a question that has no semantic match to any blog chunk (e.g., "what is dark matter?"), Vectorize returns empty results.

```
const results = await env.VECTORIZE_INDEX.query(questionEmbedding.data[0], { topK: 5 }); if (results.length === 0) {
return new Response(JSON.stringify({ answer: "I don't have information about that topic. Try asking about Cloudflare,
Workers, Durable Objects, or R2.", sources: [] }), { headers: { "Content-Type": "application/json" } }); }
```

Always handle the empty result case.

5. AI Inference Has Rate Limits

Workers AI on a free account has rate limits (reasonable for a blog feature, but be aware):

- ~1,000 requests per day for embeddings
- ~1,000 requests per day for text generation

Cached requests don't count toward the limit (they're served from AI Gateway cache).

If you hit limits, you'll get a 429 error. The fix is a Cloudflare paid plan (Workers or Workers AI add-on).

File Structure After Phase 4

```
SaltWaterBRC/ ■■■ site/ # Pages project ■ ■■■ functions/ ■ ■ ■■■ api/ ■ ■ ■■■ status.js # Phase 1: Site metadata ■ ■ ■■■ edge-info.js # Phase 1: Request inspector ■ ■■■ ask.html # Phase 4: Ask This Blog UI ■ ■■■ blog/ ■ ■ ■■■ why-im-building-on-cloudflare.html ■ ■ ■■■ the-roadmap.html ■ ■ ■■■ ... ■ ■■■ index.html ■ ■■■ wrangler.toml # Pages config ■ ■■■ counter-worker/ # Phase 2: Standalone Worker ■ ■■■ src/index.js # VisitorCounter DO + fetch handler ■ ■■■ wrangler.toml ■ ■■■ package.json ■ ■■■ ask-worker/ # Phase 4: Standalone Worker (NEW) ■ ■■■ src/ ■ ■ ■■■ index.js # RAG: embed, search, generate ■ ■■■ wrangler.toml # Bindings: AI, Vectorize, AI Gateway ■ ■■■ package.json ■ ■■■ Drafts/ # Blog post drafts ■■■ Training/ # Training docs ■ ■■■ Phase1-Pages-Deployment.md ■ ■■■ Phase2-Workers-REST-API.md ■ ■■■ Phase3-Workers-DO-R2.md ■ ■■■ Phase4-WorkersAI-Vectorize-AIGateway.md # ← This file ■■■ PROJECT-BRIEF.md # Master roadmap ■■■ CLAUDE.md # Dev ops guide
```

Key Lessons

1. **Workers AI is edge inference** — models run in your regional data center, not the cloud. Fast, private, cost-effective.

2. **Vectorize is the retrieval layer** — it finds relevant content by meaning (semantic search), not keywords. The foundation of RAG.

3. **AI Gateway is observability** — log, cache, rate limit, and analyze every AI request in one place. You wouldn't run a database without monitoring. Don't run AI without it either.

4. **Standalone Workers for complex bindings** — Pages Functions with AI + Vectorize + AI Gateway caused deployment issues. Standalone Worker (ask-worker) called from frontend works reliably.

5. **RAG = retrieval + generation** — don't answer from training data. Answer from context (your docs). More accurate, no hallucination, fully controlled.

6. **Chunking matters** — one big vector is too coarse. Many small vectors means more precise search results. 14 blog chunks is right for this use case.

7. **Caching cuts costs dramatically** — duplicate questions hit AI Gateway cache (instant, free). In real-world usage, 30-50% of questions are repeats.

Going to Production

If you were to scale "Ask This Blog" to thousands of daily questions:

1. **Monitor token usage in AI Gateway** — set up alerts if usage spikes 2. **Implement rate limiting** — AI Gateway built-in, or custom logic 3. **Add analytics** — track which questions get asked most 4. **Improve chunking** — 14 chunks work for a small blog, but a large knowledge base needs 100+ chunks with better semantic boundaries 5. **Add feedback loop** — users rate answers ("was this helpful?") to improve Vectorize indexing and context selection 6. **Consider context expansion** — instead of top 5 chunks, experiment with top 3, top 10, and measure answer quality 7. **Add regeneration** — if answer quality drops, let users click "try again" to regenerate with different context

The Pitch to Customers