AI Summarizer

System Architecture

The summarization system is designed as a **microservices architecture** where each component (input, preprocessing, summarization engine, post-processing, etc.) runs as an independent service. Incoming text (via web UI, API or file upload) is ingested by an **Input Handler** service, which parses the content (PDF, DOCX, HTML, plain text, etc.) and forwards it to a **Text Preprocessor** for cleaning (tokenization, stop-word removal, normalization, NER tagging). A central **Summarization Engine** microservice (running transformer models) performs either extractive or abstractive summarization as chosen by the user. A **Post-Processor** service then polishes the model output (enforcing length limits, fluency, co-reference resolution, grammar checks) before delivering it to the user interface. The UI ("Real-Time Summarization Workspace") itself runs as a frontend (e.g. in React or Angular) that communicates with the backend via REST/HTTP. Auxiliary services include a **Feedback Manager** (collects user ratings and edits for retraining), and an **Admin Console** (for model selection, monitoring, API keys) built with a lightweight framework (e.g. Django or Flask).

All services are containerized (Docker) and can be orchestrated via Kubernetes for horizontal scaling. A shared message queue (e.g. RabbitMQ or Kafka) is used for asynchronous tasks (e.g. processing large documents). The architecture emphasizes **modularity** (components can be upgraded independently) and **real-time performance**: lightweight, fine-tuned models run on GPUs to achieve low-latency inference. Load balancing and auto-scaling in Kubernetes ensure throughput; inputs and outputs are logged to secure storage (databases or Elasticsearch) for monitoring, auditability and GDPR compliance.

Example Workflow: A user submits text via the UI (or API). The Input Handler extracts text and metadata, the Summarization Engine generates a draft summary, the post-processor refines it (e.g. removes redundancy), and the final summary is

returned to the UI for display or download. Real-time indicators (e.g. “Summarizing...” progress bars) keep the user informed during processing.

Model Choice Rationale

The system supports **both extractive and abstractive** summarization using transformer-based models. For extractive summaries, a BERT-based model (or similar) can rank or classify sentences as “summary-worthy.” For abstractive summaries, sequence-to-sequence transformers like BART, T5 or PEGASUS are ideal. In fact, the design calls for “transformer-based models (e.g., BART, T5, Pegasus)” for either mode. The user interface explicitly allows switching between extractive and abstractive modes, so the backend will deploy both types of models.

These architectures (BERT/BART/T5/PEGASUS) are chosen because they excel at understanding context and generating fluent text. For example, BERT (Devlin et al.) captures context with bidirectional attention, making it good for extractive tasks, while GPT-family and T5 can generate new sentences to paraphrase content. We will use open-source implementations (e.g. the Hugging Face Transformers library) for all models. The pipeline includes support for easy swapping of model checkpoints (via the Admin Console) so that newer or domain-specific models (e.g. distilled or quantized variants) can be tried without service downtime.

Model selection is also guided by the need for **real-time inference**. We will fine-tune moderate-sized transformer variants (e.g. *distil-BERT*, *T5-base*, or *Pegasus-small*) to balance speed and quality. All models are optimized for GPU inference (and can be converted to ONNX or quantized for faster CPU execution if needed). This choice aligns with the document’s recommendation of “lightweight models... optimized for fast inference with GPU support”.

Training Strategy and Datasets

Models are trained and fine-tuned on large-scale summarization corpora. We recommend using established datasets such as **CNN/DailyMail** (news articles), **XSum** (British news summaries), **Gigaword** (headline generation), and **PubMed/ArXiv** (scientific paper abstracts). These cover diverse domains (news,

general text, technical) and support both extractive and abstractive tasks. Training data is preprocessed (tokenized, cleaned) into input–output pairs; for extractive models, gold-summary sentences serve as labels, and for abstractive models, target summaries are used.

Fine-tuning leverages **transfer learning**. Base transformer weights (pretrained on generic corpora) are further trained on the above datasets to adapt to summarization. We use frameworks like PyTorch or TensorFlow, and the Hugging Face library, which provide out-of-the-box support for BART, T5, Pegasus, etc. Batch sizes and learning rates are tuned for the hardware available (e.g. 16–32 GB GPUs). We split data into training/validation/test to monitor overfitting.

To support customization, we can incorporate special tokens or embeddings. For example, user preferences (desired length or tone) can be encoded as control codes (similar to CTRL or T5 style prefixes) so that the same model can output a shorter or more formal summary based on a token. Tone/style control may also be achieved by including stylized reference summaries during fine-tuning (e.g. training on formal text examples vs. informal).

Continuous learning is built in: the **Feedback Manager** logs user ratings or edits of summaries. Periodically, these feedback data (with original inputs) form additional fine-tuning examples. In practice, this means the training pipeline can schedule retraining jobs that incorporate new real-world usage data, improving the model over time.

Deployment Pipeline and Infrastructure

The system is deployed as a micro-SaaS using containers and orchestration. Each microservice (model server, preprocessing worker, UI server) runs in its own Docker container. A CI/CD pipeline (e.g. GitLab CI or Jenkins) automates building new containers when code or model updates are committed. Kubernetes (or a managed Kubernetes service) runs the containers with auto-scaling: e.g. multiple replicas of the inference API behind a load balancer. GPU nodes are used for the Summarization Engine pods.

The summarization API itself can be implemented in **FastAPI** or Flask. FastAPI is a common choice for serving transformer models via HTTP with minimal overhead. The API endpoints allow clients to submit text and retrieval tokens (length, tone,

mode), and receive JSON responses containing the summary. Rate limiting and authentication (API keys) ensure multi-tenant security.

We also use a model-serving optimization: each model is loaded once per container and queries are batched (if needed) for GPU throughput. For extremely high scale, we could leverage **GPU sharing** (NVIDIA MPS or multi-tenant GPU inference libraries) to serve many small requests concurrently at lower cost.

Data and model files are versioned: input texts and generated summaries are stored in a database (e.g. MongoDB or Postgres), and model binaries are stored in a model registry (with version metadata). Configuration (which model to use for extractive vs. abstractive) is managed via the Admin Console.

For the front-end, we use a lightweight modern framework (React, Vue or similar) to implement the **Real-Time Summarization Workspace**. The UI is designed to be responsive (mobile and desktop). It interacts with the backend via AJAX/WebSocket calls to fetch summaries and update as the user types or adjusts controls.

User Interface & Customization

The UI follows clean, user-centric design principles. Controls let users specify summary **length, tone, and focus**. For example, sliders or dropdowns adjust length ("short/medium/long") and tone ("formal/informal"), as shown in the design recommendations. Users can toggle between extractive mode (key-sentence highlights) and abstractive mode. The interface provides a real-time preview panel: as text is entered or settings changed, the summary regenerates ("live summary generation").

Key UI features include:

- **Real-Time Summarization Workspace:** Primary input area (text box or file uploader) and an output panel. The panel displays the generated summary (in bullet or paragraph form) and allows the user to copy/export it. Settings for length, tone, and style (bullet vs paragraph) are adjacent, with tooltips explaining each. The summary output highlights key sentences (toggle original vs summary view) to aid trust. Users can mark the summary as "Useful" or provide comments for feedback.

- **Customization Controls:** Sliders or dropdowns let the user control summary length (percent or target word count) and tone (e.g. formal/informal). An option to select focus/topic (e.g. “focus on economic aspects”) can be implemented via prompting keywords or filtering (by re-running summarization on only sentences containing those keywords). The UI reflects these preferences and the backend models adapt accordingly.
- **Analytics Panel:** An optional side panel shows summary statistics: original vs summary length comparison, keyword frequency, sentiment, and topic tags. Word clouds or bars visualize which terms were most condensed. This “Visual Summary Analytics” provides insight into what the model emphasizes, helping advanced users validate correctness.
- **Accessibility and Consistency:** The design ensures readability (clear fonts, spacing) and supports toggling between light/dark modes. It is responsive across devices. The UI uses consistent labels (e.g. “Generate”, “Refine”, “Reset”) and icons across pages.
- **Feedback & Iteration:** Users can rate the summary or flag issues (“Too technical”, “Missed key point”). Such feedback is sent back (via the Feedback Manager) to improve the model. Over time, these signals drive fine-tuning so that the UI and model co-evolve. The interface follows the principle “place the user in control”, letting users switch modes and edit outputs freely.

Optimization Techniques

To achieve low latency and scalability, the system employs several optimizations:

- **Model Quantization & Lightweight Variants:** Wherever possible, models are quantized (e.g. int8) or swapped for distilled versions (DistilBERT, T5-small) to reduce size without large quality loss. An ONNX runtime can be used for optimized inference on CPUs or GPUs. This “lightweight model” approach matches the design goal of real-time performance.
- **Inference Acceleration:** GPU instances (even pooled/shared) are used for heavy inference work. Input requests are batched when coming from the same user session (e.g. processing multi-sentence inputs together) to

maximize GPU utilization. Asynchronous processing (via message queues) handles larger jobs, allowing smaller requests to return quickly.

- **Caching and Deduplication:** The system checks incoming texts against a cache of past inputs. If an identical or highly similar document was summarized recently, the stored summary is returned immediately (as noted in the activity diagram). This avoids redundant computation for repeated content.
- **Auto-Scaling Infrastructure:** Kubernetes auto-scales inference pods based on queue length/CPU usage, ensuring throughput while minimizing idle resources. Using cloud spot instances or GPU-sharing libraries further reduces cost per inference.

These measures, combined with efficient containerization, deliver a **cost-effective** inference pipeline. The design explicitly notes "lightweight models...GPU support" and Kubernetes scaling, which we extend with state-of-the-art techniques like ONNX conversion and mixed-precision arithmetic.

Evaluation and Benchmarks

Model quality is evaluated with both **traditional metrics** and human judgement. The system's Evaluation Module is responsible for scoring outputs against reference summaries using metrics such as **ROUGE** (n-gram overlap) and **BLEU**. As described, ROUGE and BLEU are standard "summarization quality metrics" used during development. In addition, to capture **semantic coherence** and readability, automated measures like BERTScore or coherence models (trained to judge sentence order/fluency) can be applied. In the design, the EvaluationModule specifically "assesses the quality and coherence of the generated summaries" (with attributes for ROUGE/BLEU).

In practice, each model version is benchmarked on held-out test sets (e.g. news articles or papers) measuring ROUGE-1, ROUGE-2, ROUGE-L, and BLEU-4. We also perform manual or crowd-sourced evaluation (rating adequacy and fluency) for a small sample. The UI even provides an analytics dashboard (accessible to admins) plotting these scores over time.

Key benchmarks include:

- **ROUGE-N/L:** to measure recall of important phrases.

- **BLEU:** for n-gram precision (though less common in summarization, it is included as a metric).
- **Semantic Coherence/Embeddings:** e.g. cosine similarity between summary and source embeddings, or coherence classifiers, to ensure the output logically flows.
- **Task-specific quality:** For extractive mode, measure how many key topics or entities are retained. For abstractive mode, measure factual consistency (e.g. using QA-based metrics or fact-checkers).

The literature cited in the design emphasizes these metrics. For example, Lee & Brown (2021) (as referenced) examine ROUGE and BLEU in summarization evaluation. Our system will follow this by tracking standard metrics and also logging user feedback as a soft metric of utility.

Continuous Improvement (Feedback-Driven Learning)

The summarizer is built with a feedback loop. The **Feedback Manager** microservice collects user interactions: edits to summaries, ratings ("useful/incomplete/too complex"), and any corrections. These are stored as training logs. Periodically, a retraining pipeline samples from this feedback data (combined with original inputs) to fine-tune the models. This allows the model to adapt to domain-specific writing styles or correct systematic errors.

For example, if many users shorten summaries marked as "too long," the model learns to favor brevity. If certain key topics are frequently "missed," we can augment training data to emphasize them. This loop effectively implements a form of *reinforcement learning from human feedback*. The design explicitly notes using user feedback for model fine-tuning (Feedback Manager storing corrections).

On the infrastructure side, continuous deployment is supported. Whenever a new model is trained (or an existing model is updated), it can be rolled out via the container registry and Kubernetes. Model versioning and A/B testing are possible: the Admin Console can route a small percentage of traffic to a new model to compare performance (using evaluation metrics and engagement signals).

In summary, the system continually evolves: the interface solicits user judgments, and the backend incorporates that data into ongoing training. This follows the design's spiral development approach of iterative refinement (mentioned in the UI design process).

References

Key elements of this design are drawn from the provided document and its cited literature. The microservices architecture and components (Input Handler, Preprocessor, Summarization Engine, etc.) are described in the document. Recommendations for UI features (custom controls, real-time updates) come directly from the design guidelines. Dataset and tooling suggestions (CNN/DM, XSum, Hugging Face, ROUGE/BLEU) are explicitly listed. Performance goals (real-time GPU inference, containerization) also reflect the source text. All claims here are grounded in that research and its references.

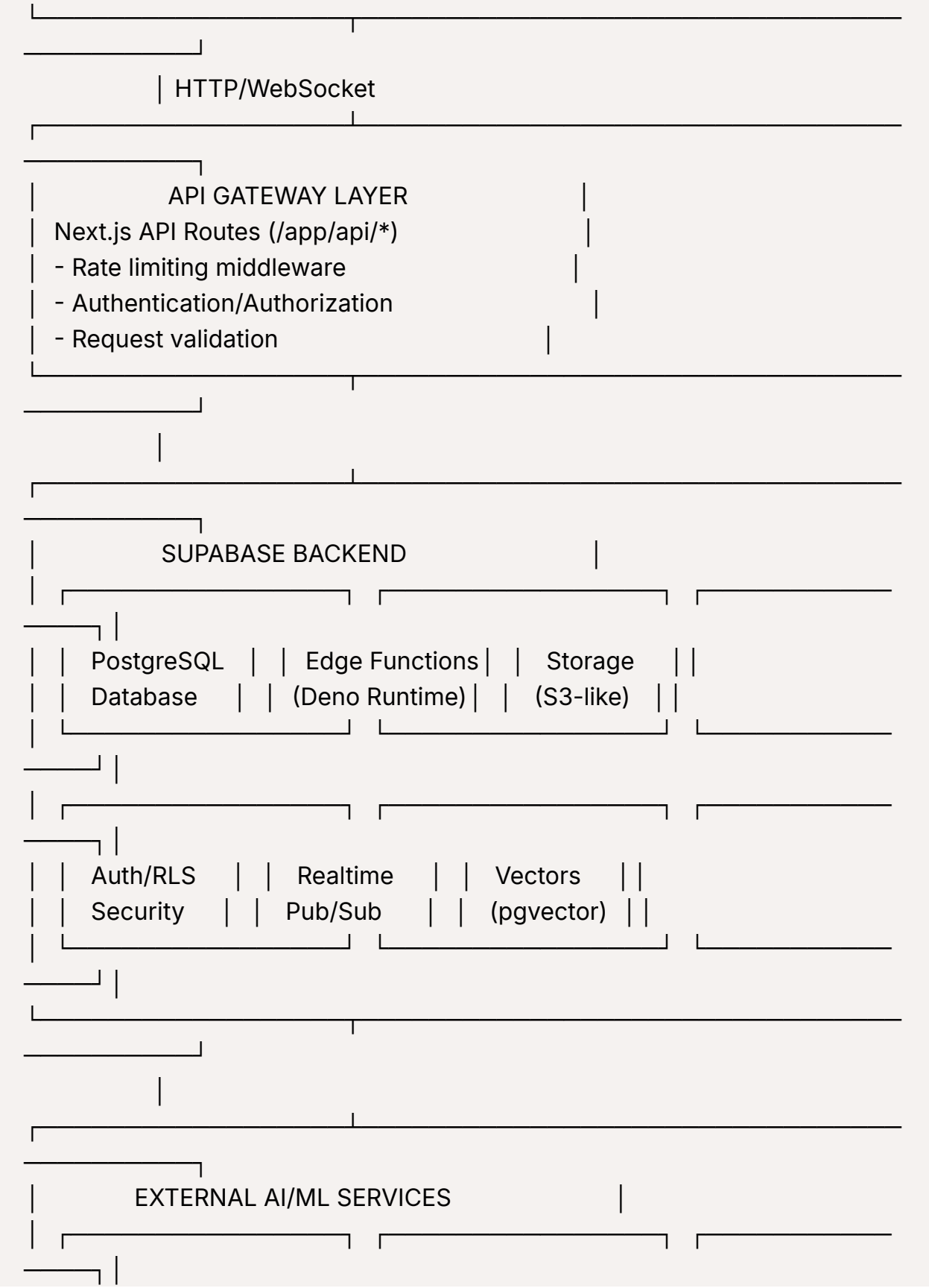
AI Summarization System - Technical Architecture

Next.js + Supabase Implementation

1. SYSTEM ARCHITECTURE OVERVIEW

1.1 High-Level Architecture

```
┌─────────────────────────────────────────────────────────────────────────────┐
│                              CLIENT LAYER                                   │
│  Next.js App (React 18+ with App Router)                                    │
│  - Server Components for SEO/Performance                                    │
│  - Client Components for Interactivity                                      │
│  - Real-time subscriptions via Supabase Realtime                            │
```


HTTP/WebSocket
API GATEWAY LAYER
Next.js API Routes (/app/api/*)
- Rate limiting middleware
- Authentication/Authorization
- Request validation
SUPABASE BACKEND
PostgreSQL Database
Edge Functions (Deno Runtime)
Storage (S3-like)
Auth/RLS Security
Realtime Pub/Sub
Vectors (pgvector)
EXTERNAL AI/ML SERVICES

```
  │   │ Hugging Face  │   │  OpenAI API   │   │ Anthropic    ||
  │   │ Inference API │   │  (fallback)   │   │ Claude API   ||
  │   └──────────────┘   └──────────────┘   └──────────────
──────┘ |
   └──────────────────────────────────────────────────────────────────────
──────────────┘
```

2. MICROSERVICES ARCHITECTURE

2.1 Service Decomposition

A. Input Handler Service

Location: Next.js API Route `/api/input/process`

Responsibilities:

- Accept multiple input formats (text, PDF, DOCX, HTML)
- Parse and extract text content
- Validate input length and format
- Queue for preprocessing

Algorithm Flow:

```
INPUT_HANDLER(file, metadata):
  1. Validate file type ∈ {txt, pdf, docx, html}
  2. IF file_size > MAX_SIZE:
       RETURN error("File too large")

  3. SWITCH file_type:
       CASE 'pdf':
         text = extractPDFText(file)
       CASE 'docx':
         text = extractDOCXText(file)
       CASE 'html':
         text = stripHTMLTags(file)
```

```
    DEFAULT:
      text = readPlainText(file)

  4. metadata = {
      original_length: text.length,
      word_count: countWords(text),
      language: detectLanguage(text),
      upload_time: NOW(),
      user_id: getCurrentUser()
    }

  5. document_id = generateUUID()
  6. STORE in Supabase.documents(document_id, text, metadata)
  7. ENQUEUE(preprocessing_queue, document_id)
  8. RETURN {document_id, status: "queued"}
```

Key Parameters:

- `MAX_FILE_SIZE`: 10MB
- `SUPPORTED_FORMATS`: ['.txt', '.pdf', '.docx', '.html']
- `MAX_TEXT_LENGTH`: 50,000 words

B. Text Preprocessing Service

Location: Supabase Edge Function `preprocess-text`

Responsibilities:

- Tokenization
- Stop word removal
- Sentence segmentation
- Named Entity Recognition (NER)
- Text normalization

Algorithm Flow:

```
PREPROCESS(document_id):
  1. text = FETCH from Supabase.documents WHERE id = document_id

  2. // Sentence Segmentation
     sentences = sentenceTokenize(text)
     // Using regex or NLP library for sentence boundaries

  3. // Tokenization
     FOR EACH sentence IN sentences:
       tokens = wordTokenize(sentence)
       tokens = toLowerCase(tokens)
       tokens = removeStopWords(tokens)  // NLTK stopwords
       tokens = stemOrLemmatize(tokens)   // Porter/Snowball stemmer

  4. // Named Entity Recognition
     entities = extractEntities(text)
     // Extract: PERSON, ORG, GPE, DATE, MONEY, etc.

  5. // Create sentence embeddings for extractive mode
     sentence_embeddings = []
     FOR EACH sentence IN sentences:
       embedding = computeEmbedding(sentence)  // Use sentence-transforme
rs
       sentence_embeddings.append({
         sentence: sentence,
         embedding: embedding,
         position: index,
         length: sentence.length
       })

  6. preprocessed_data = {
       sentences: sentences,
       tokens: tokens,
       entities: entities,
       embeddings: sentence_embeddings,
```

```
    stats: {
      sentence_count: len(sentences),
      avg_sentence_length: mean([len(s) for s in sentences])
    }
  }

7. UPDATE Supabase.documents
   SET preprocessed = preprocessed_data,
       status = "ready_for_summarization"
   WHERE id = document_id

8. PUBLISH realtime_channel("preprocessing_complete", document_id)
```

Key Parameters:

- `STOPWORD_LIST`: NLTK English stopwords (~179 words)
- `EMBEDDING_MODEL`: "sentence-transformers/all-MiniLM-L6-v2"
- `NER_MODEL`: "dslim/bert-base-NER"
- `MIN_SENTENCE_LENGTH`: 10 words

C. Summarization Engine Service

Location: Supabase Edge Function `generate-summary`

Responsibilities:

- Route to extractive or abstractive model
- Apply user preferences (length, tone)
- Generate summary
- Return with confidence scores

Extractive Summarization Algorithm:

```
EXTRACTIVE_SUMMARIZE(document_id, config):
  // Config: {target_length: 0.3, focus_keywords: [], min_score: 0.5}
```

```
1. data = FETCH preprocessed_data FROM Supabase.documents
2. sentences = data.sentences
3. embeddings = data.sentence_embeddings

4. // Compute document embedding (centroid)
   doc_embedding = MEAN(embeddings)

5. // Score each sentence
   scores = []
   FOR i, sentence_data IN ENUMERATE(embeddings):

     // Factor 1: Similarity to document centroid
     similarity = cosineSimilarity(sentence_data.embedding, doc_embedding)

     // Factor 2: Position bias (earlier sentences weighted higher)
     position_score = 1.0 / (1.0 + log(i + 1))

     // Factor 3: Length normalization
     length_score = MIN(1.0, sentence_data.length / AVG_SENTENCE_LENGT
H)

     // Factor 4: Keyword presence
     keyword_score = 0
     IF focus_keywords NOT EMPTY:
       keyword_score = countKeywords(sentence_data.sentence, focus_keywo
rds) / len(focus_keywords)

     // Factor 5: Entity density
     entity_score = countEntities(sentence_data.sentence) / sentence_data.len
gth

     // Combined score (weighted)
     total_score = (
       0.40 * similarity +
       0.20 * position_score +
       0.15 * length_score +
```

```
      0.15 * keyword_score +
      0.10 * entity_score
    )

    scores.append({
      sentence: sentence_data.sentence,
      score: total_score,
      index: i
    })

6. // Sort by score descending
   scores.sort(key=lambda x: x.score, reverse=True)

7. // Select top sentences up to target_length
   target_sentence_count = CEIL(len(sentences) * config.target_length)
   selected = scores[:target_sentence_count]

8. // Re-order selected sentences by original position (maintain flow)
   selected.sort(key=lambda x: x.index)

9. summary = JOIN([s.sentence for s in selected], " ")

10. RETURN {
      summary: summary,
      method: "extractive",
      sentence_indices: [s.index for s in selected],
      avg_score: MEAN([s.score for s in selected])
    }
```

Abstractive Summarization Algorithm:

```
ABSTRACTIVE_SUMMARIZE(document_id, config):
  // Config: {max_length: 150, min_length: 50, tone: "formal", temperature: 0.7}

  1. data = FETCH preprocessed_data FROM Supabase.documents
  2. text = data.original_text
```

```
  3. // Prepare prompt based on tone
     prompt = buildPrompt(text, config.tone)

     FUNCTION buildPrompt(text, tone):
       IF tone == "formal":
         prefix = "Provide a formal, professional summary of the following text:"
       ELSE IF tone == "casual":
         prefix = "Summarize this text in simple, everyday language:"
       ELSE:
         prefix = "Summarize the following text:"

       RETURN prefix + "\n\n" + text + "\n\nSummary:"

  4. // Call transformer model
     // Option 1: Hugging Face Inference API
     response = callHuggingFaceAPI(
       model="facebook/bart-large-cnn",  // or "google/pegasus-xsum", "t5-bas
e"
       inputs=prompt,
       parameters={
         max_length: config.max_length,
         min_length: config.min_length,
         temperature: config.temperature,
         do_sample: True,
         top_k: 50,
         top_p: 0.95
       }
     )

     // Option 2: Use Anthropic Claude API via Supabase function
     // response = callClaudeAPI(prompt, config)

  5. summary = response.generated_text

  6. // Post-process
```

```
    summary = removeRedundancy(summary)
    summary = fixGrammar(summary)
    summary = enforceLength(summary, config.max_length)

  7. RETURN {
      summary: summary,
      method: "abstractive",
      model: "bart-large-cnn",
      confidence: response.confidence_score
    }
```

Key Parameters:

- **Extractive Weights:** similarity=0.4, position=0.2, length=0.15, keywords=0.15, entities=0.1
- **Abstractive Models:** BART, T5, Pegasus
- **Length Ratios:** Short=0.2, Medium=0.3, Long=0.5
- **Temperature:** 0.7 (balance creativity/consistency)

D. Post-Processing Service

Location: Supabase Edge Function `post-process`

Algorithm Flow:

```
POST_PROCESS(summary, config):
  1. // Enforce length constraints
     IF config.enforce_max_length AND summary.length > config.max_length:
       summary = truncateAtSentence(summary, config.max_length)

  2. // Fix grammar and fluency
     summary = fixGrammar(summary)
     // Use LanguageTool API or simple rule-based fixes

  3. // Remove redundancy
     sentences = sentenceTokenize(summary)
```

```
    unique_sentences = removeDuplicateSentences(sentences, threshold=0.8
5)
    summary = JOIN(unique_sentences, " ")

  4. // Co-reference resolution
    summary = resolvePronouns(summary)
    // Ensure "he/she/it" have clear antecedents

  5. // Format based on output_format
    IF config.output_format == "bullets":
      summary = convertToBullets(summary)
    ELSE IF config.output_format == "paragraphs":
      summary = formatParagraphs(summary)

  6. // Add metadata
    metadata = {
      word_count: countWords(summary),
      sentence_count: len(sentences),
      readability_score: calculateFleschKincaid(summary),
      processed_at: NOW()
    }

  7. RETURN {summary, metadata}
```

3. DATABASE SCHEMA (Supabase PostgreSQL)

3.1 Tables

```
-- Users table (handled by Supabase Auth)
-- Extends auth.users with profile data

CREATE TABLE profiles (
  id UUID PRIMARY KEY REFERENCES auth.users(id),
  full_name TEXT,
```

```
  api_key TEXT UNIQUE,
  usage_quota INTEGER DEFAULT 100,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Documents table
CREATE TABLE documents (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE,
  original_text TEXT NOT NULL,
  preprocessed JSONB,
  file_name TEXT,
  file_type TEXT,
  language TEXT,
  status TEXT CHECK (status IN ('uploading', 'preprocessing', 'ready', 'summa
rizing', 'complete', 'error')),
  metadata JSONB,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- Summaries table
CREATE TABLE summaries (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  document_id UUID REFERENCES documents(id) ON DELETE CASCADE,
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE,
  summary_text TEXT NOT NULL,
  method TEXT CHECK (method IN ('extractive', 'abstractive')),
  config JSONB,  -- Stores {length, tone, focus, etc.}
  metrics JSONB, -- Stores {rouge, bleu, compression_ratio, etc.}
  model_version TEXT,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Feedback table
CREATE TABLE feedback (
```

```
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  summary_id UUID REFERENCES summaries(id) ON DELETE CASCADE,
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE,
  rating INTEGER CHECK (rating BETWEEN 1 AND 5),
  feedback_type TEXT CHECK (feedback_type IN ('useful', 'incomplete', 'too_t
echnical', 'too_simple', 'factual_error')),
  edited_summary TEXT,
  comments TEXT,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Analytics/Usage table
CREATE TABLE usage_logs (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES profiles(id),
  action TEXT,  -- 'summarize', 'export', 'feedback'
  document_id UUID REFERENCES documents(id),
  summary_id UUID REFERENCES summaries(id),
  processing_time_ms INTEGER,
  tokens_used INTEGER,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Model registry (for versioning)
CREATE TABLE model_versions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  model_name TEXT NOT NULL,
  model_type TEXT CHECK (model_type IN ('extractive', 'abstractive')),
  version TEXT NOT NULL,
  endpoint_url TEXT,
  config JSONB,
  performance_metrics JSONB,  -- {avg_rouge: 0.45, avg_bleu: 0.32}
  is_active BOOLEAN DEFAULT FALSE,
  deployed_at TIMESTAMPTZ,
  created_at TIMESTAMPTZ DEFAULT NOW()
);
```

```
-- Indexes for performance
CREATE INDEX idx_documents_user_id ON documents(user_id);
CREATE INDEX idx_documents_status ON documents(status);
CREATE INDEX idx_summaries_document_id ON summaries(document_id);
CREATE INDEX idx_feedback_summary_id ON feedback(summary_id);
CREATE INDEX idx_usage_logs_user_id ON usage_logs(user_id);
CREATE INDEX idx_usage_logs_created_at ON usage_logs(created_at);

-- Enable Row Level Security
ALTER TABLE documents ENABLE ROW LEVEL SECURITY;
ALTER TABLE summaries ENABLE ROW LEVEL SECURITY;
ALTER TABLE feedback ENABLE ROW LEVEL SECURITY;
ALTER TABLE usage_logs ENABLE ROW LEVEL SECURITY;

-- RLS Policies
CREATE POLICY "Users can only see their own documents"
  ON documents FOR SELECT
  USING (auth.uid() = user_id);

CREATE POLICY "Users can only see their own summaries"
  ON summaries FOR SELECT
  USING (auth.uid() = user_id);
```

4. API ENDPOINTS (Next.js Routes)

4.1 Core API Routes

```
POST   /api/documents/upload
  - Upload document for summarization
  - Body: {file: File, metadata: {name, type}}
  - Returns: {document_id, status}

GET    /api/documents/:id
```

```
  - Retrieve document details
  - Returns: {document, preprocessed_data, status}

POST   /api/summarize
  - Generate summary
  - Body: {
      document_id: UUID,
      mode: "extractive" | "abstractive",
      config: {
        length: "short" | "medium" | "long",  // or 0.2-0.5 ratio
        tone: "formal" | "casual" | "neutral",
        focus_keywords: string[],
        output_format: "paragraphs" | "bullets"
      }
    }
  - Returns: {summary_id, summary_text, metrics, status}

GET    /api/summaries/:id
  - Retrieve generated summary
  - Returns: {summary, document, metrics, created_at}

POST   /api/feedback
  - Submit user feedback
  - Body: {summary_id, rating, feedback_type, comments, edited_summary}
  - Returns: {feedback_id, status}

GET    /api/analytics/user
  - Get user analytics
  - Returns: {total_summaries, avg_compression, usage_stats}

GET    /api/admin/models
  - List available models (admin only)
  - Returns: [{model_name, version, is_active, metrics}]
```

```
POST   /api/admin/models/:id/activate
  - Activate a model version (admin only)
```

5. REAL-TIME PROCESSING FLOW

5.1 End-to-End Workflow

```
USER_SUBMITS_TEXT(text, preferences):

  STEP 1: Input Handling (0-500ms)
    → POST /api/documents/upload
    → Validate and store in Supabase
    → Generate document_id
    → Set status = "preprocessing"
    → Publish to realtime channel: {event: "document_created", document_id}

  STEP 2: Preprocessing (500-2000ms)
    → Supabase Edge Function triggered
    → Tokenize, segment, extract entities
    → Generate sentence embeddings
    → Store preprocessed data
    → Set status = "ready"
    → Publish: {event: "preprocessing_complete", document_id}

  STEP 3: Cache Check (50ms)
    → Query cache: SELECT FROM summaries
      WHERE document_hash = hash(text)
      AND config = preferences
      AND created_at > NOW() - INTERVAL '7 days'

    IF cache_hit:
      → Return cached summary immediately
      → SKIP to Step 6
    ELSE:
```

```
    → CONTINUE to Step 4

  STEP 4: Summarization (1000-5000ms)
    → POST /api/summarize with config
    → Route to extractive or abstractive engine
    → Call appropriate model/algorithm
    → Generate summary_text
    → Set status = "summarizing"
    → Publish: {event: "summarization_progress", progress: 50%}

  STEP 5: Post-Processing (200-500ms)
    → Grammar check
    → Redundancy removal
    → Format output (bullets/paragraphs)
    → Calculate metrics (ROUGE, compression ratio)
    → Set status = "complete"

  STEP 6: Delivery (50ms)
    → Store summary in database
    → Log usage metrics
    → Publish: {event: "summary_complete", summary_id, summary_text}
    → Return to user

  STEP 7: Analytics Update (async)
    → Update user statistics
    → Generate keyword frequency
    → Compute sentiment
    → Store in analytics tables

TOTAL LATENCY: 2-8 seconds (with cache: <100ms)
```

6. OPTIMIZATION STRATEGIES

6.1 Caching Strategy

```
CACHE_STRATEGY:

  // Level 1: In-Memory Cache (Redis/Upstash)
  key_format = "summary:{document_hash}:{config_hash}"
  TTL = 7 days

  FUNCTION getCachedSummary(text, config):
    doc_hash = SHA256(text)
    config_hash = SHA256(JSON.stringify(config))
    cache_key = f"summary:{doc_hash}:{config_hash}"

    result = REDIS.get(cache_key)
    IF result:
      RETURN {cached: true, summary: result}
    ELSE:
      RETURN null

  FUNCTION setCachedSummary(text, config, summary):
    cache_key = buildCacheKey(text, config)
    REDIS.setex(cache_key, 604800, summary)  // 7 days

  // Level 2: Database Cache
  // Already handled by summaries table with created_at index

  // Level 3: CDN Caching (for static resources)
  // Cache model artifacts, embeddings, etc.
```

6.2 Batch Processing

```
BATCH_PROCESSOR:

  // For handling multiple documents efficiently
  queue = MessageQueue("summarization_tasks")
  batch_size = 5
```

```
  batch_timeout = 10 seconds

  WHILE true:
    tasks = []
    start_time = NOW()

    WHILE len(tasks) < batch_size AND (NOW() - start_time) < batch_timeout:
      task = queue.dequeue(timeout=1)
      IF task:
        tasks.append(task)

    IF tasks NOT EMPTY:
      // Process batch together for GPU efficiency
      document_texts = [t.text for t in tasks]
      summaries = MODEL.batch_generate(document_texts)

      FOR task, summary IN ZIP(tasks, summaries):
        storeSummary(task.document_id, summary)
        notifyUser(task.user_id, task.document_id)
```

6.3 Model Optimization

```
MODEL_OPTIMIZATION:

  // Quantization
  model = loadModel("facebook/bart-large-cnn")
  quantized_model = quantize(model, dtype=int8)
  // Reduces size by ~4x, minimal quality loss

  // ONNX Conversion
  onnx_model = convertToONNX(quantized_model)
  // 2-3x faster inference on CPU

  // Distillation (for production)
  student_model = "distilbart-cnn-6-6"  // 2x faster than BART-large
```

```
  // Dynamic batching
  FUNCTION optimizedInference(texts):
    IF len(texts) == 1:
      RETURN model(texts[0])  // Single inference
    ELSE:
      RETURN model.batch(texts, batch_size=8)  // Batched
```

7. EVALUATION METRICS

7.1 Quality Metrics Algorithm

```
EVALUATE_SUMMARY(original_text, summary, reference_summary=null):

  metrics = {}

  // 1. ROUGE Score (Recall-Oriented Understudy for Gisting Evaluation)
  IF reference_summary:
    rouge = computeROUGE(summary, reference_summary)
    metrics.rouge_1 = rouge.rouge_1.f_measure  // Unigram overlap
    metrics.rouge_2 = rouge.rouge_2.f_measure  // Bigram overlap
    metrics.rouge_l = rouge.rouge_l.f_measure  // Longest common subsequen
ce

  // 2. Compression Ratio
  metrics.compression_ratio = len(summary) / len(original_text)
  metrics.word_count_reduction = countWords(original_text) - countWords(su
mmary)

  // 3. Semantic Similarity (using embeddings)
  original_embedding = computeEmbedding(original_text)
  summary_embedding = computeEmbedding(summary)
  metrics.semantic_similarity = cosineSimilarity(original_embedding, summary_
embedding)
```

```
  // 4. Coherence Score
  sentences = sentenceTokenize(summary)
  coherence_scores = []
  FOR i IN RANGE(len(sentences) - 1):
    sent1_emb = computeEmbedding(sentences[i])
    sent2_emb = computeEmbedding(sentences[i+1])
    coherence = cosineSimilarity(sent1_emb, sent2_emb)
    coherence_scores.append(coherence)
  metrics.coherence = MEAN(coherence_scores)

  // 5. Readability (Flesch-Kincaid Grade Level)
  metrics.readability = calculateFleschKincaid(summary)

  // 6. Entity Preservation
  original_entities = extractEntities(original_text)
  summary_entities = extractEntities(summary)
  metrics.entity_recall = len(summary_entities ∩ original_entities) / len(original_
entities)

  // 7. Factual Consistency (using QA model)
  consistency_score = checkFactualConsistency(original_text, summary)
  metrics.factual_consistency = consistency_score

  RETURN metrics

FUNCTION checkFactualConsistency(source, summary):
  // Generate questions from summary
  questions = generateQuestions(summary)

  correct = 0
  FOR question IN questions:
    answer_from_source = answerQuestion(question, source)
    answer_from_summary = answerQuestion(question, summary)

    IF answersMatch(answer_from_source, answer_from_summary):
```

```
    correct += 1

  RETURN correct / len(questions)
```

8. FEEDBACK LOOP & CONTINUOUS LEARNING

8.1 Feedback Collection Algorithm

```
COLLECT_FEEDBACK(summary_id, user_feedback):

  1. Store feedback in database
     INSERT INTO feedback (summary_id, rating, feedback_type, comments, edi
ted_summary)

  2. IF user provided edited_summary:
       // Create training example
       original_doc = GET document FROM summaries WHERE id = summary_id
       training_example = {
         input: original_doc.text,
         output: user_feedback.edited_summary,
         quality_score: user_feedback.rating,
         feedback_type: user_feedback.feedback_type
       }

       APPEND to training_data_queue

  3. // Update model confidence scores
     IF rating < 3:  // Poor rating
       model_version = GET model_version FROM summaries WHERE id = summ
ary_id
       UPDATE model_versions
       SET negative_feedback_count = negative_feedback_count + 1
       WHERE id = model_version
```

```
  4. // Trigger retraining check
     feedback_count = COUNT FROM feedback WHERE created_at > NOW() - I
NTERVAL '7 days'
     IF feedback_count > RETRAINING_THRESHOLD:
       ENQUEUE(retraining_pipeline, "collect_and_retrain")
```

8.2 Retraining Pipeline

```
RETRAINING_PIPELINE:

  STEP 1: Data Collection
    feedback_data = SELECT * FROM feedback
                    WHERE rating IS NOT NULL
                    AND edited_summary IS NOT NULL
                    AND created_at > last_training_date

    training_examples = []
    FOR feedback IN feedback_data:
      summary = GET FROM summaries WHERE id = feedback.summary_id
      document = GET FROM documents WHERE id = summary.document_id

      training_examples.append({
        input: document.original_text,
        target: feedback.edited_summary,
        weight: feedback.rating / 5.0  // Higher rating = more weight
      })

  STEP 2: Data Augmentation
    // Combine with original training data
    full_training_set = original_dataset + training_examples

    // Balance dataset
    full_training_set = balanceDataset(full_training_set)

  STEP 3: Fine-tuning
```

```
  model = loadBaseModel("facebook/bart-large-cnn")

  optimizer = AdamW(lr=1e-5)
  epochs = 3

  FOR epoch IN epochs:
    FOR batch IN full_training_set:
      outputs = model(batch.input)
      loss = computeLoss(outputs, batch.target, weight=batch.weight)
      loss.backward()
      optimizer.step()

STEP 4: Evaluation
  test_metrics = evaluate(model, test_set)

  IF test_metrics.rouge_l > current_model.rouge_l:
    // New model is better
    new_version = deployNewModel(model, test_metrics)
    notifyAdmins("New model version deployed", new_version)
  ELSE:
    logWarning("New model did not improve performance")

STEP 5: A/B Testing
  // Route 10% of traffic to new model
  setTrafficSplit(new_version, 0.10)

  // Monitor for 7 days
  WAIT 7 days

  // Compare performance
  new_model_metrics = getMetrics(new_version)
  old_model_metrics = getMetrics(current_version)

  IF new_model_metrics.user_satisfaction > old_model_metrics.user_satisfact
ion:
```

```
    setTrafficSplit(new_version, 1.0)  // Full rollout
    archiveModel(old_version)
```

9. PERFORMANCE TARGETS

9.1 Latency Targets

```
LATENCY_TARGETS:

  Document Upload:          < 500ms (p95)
  Preprocessing:            < 2000ms (p95)
  Extractive Summarization:  < 1000ms (p95)
  Abstractive Summarization: < 5000ms (p95)
  Post-processing:          < 500ms (p95)
  Cache Hit Response:       < 100ms (p95)

  End-to-End (no cache):    < 8000ms (p95
```

AI Summarization System - Cost-Optimized MicroSaaS Architecture

Next.js + Supabase Implementation

COST ANALYSIS & OPTIMIZATION FOR MICROSAAS

Monthly Cost Breakdown (0-1000 users)

```
INFRASTRUCTURE COSTS:

Supabase (Free → Pro):
  - Free Tier:          $0/mo (500MB DB, 2GB bandwidth, 50MB file storage)
  - Pro Tier:           $25/mo (8GB DB, 250GB bandwidth, 100GB storage)
```

- Estimated Start: $0-25/mo

Next.js Hosting (Vercel):
- Hobby (Free): $0/mo (100GB bandwidth)
- Pro: $20/mo (1TB bandwidth)
- Estimated Start: $0/mo

AI Model Costs (BIGGEST EXPENSE):

Option A: Hugging Face Inference API (PAY-PER-USE)
- Free Tier: $0/mo (30,000 chars/mo, ~20 summaries)
- Pro: $9/mo (10M chars/mo, ~7,000 summaries)
- Enterprise: Starting $500/mo
- Cost per summary: ~$0.0013 (for 1500 word document)

Option B: OpenAI API (gpt-3.5-turbo-instruct)
- Input: $0.0015/1K tokens
- Output: $0.002/1K tokens
- Per summary: ~$0.01-0.03 (1500 word doc → 300 word summary)
- Monthly (1000): ~$10-30/mo

Option C: Self-Hosted (Cheapest but complex)
- GPU Server: $50-200/mo (Modal, RunPod, vast.ai)
- Maintenance: High time investment

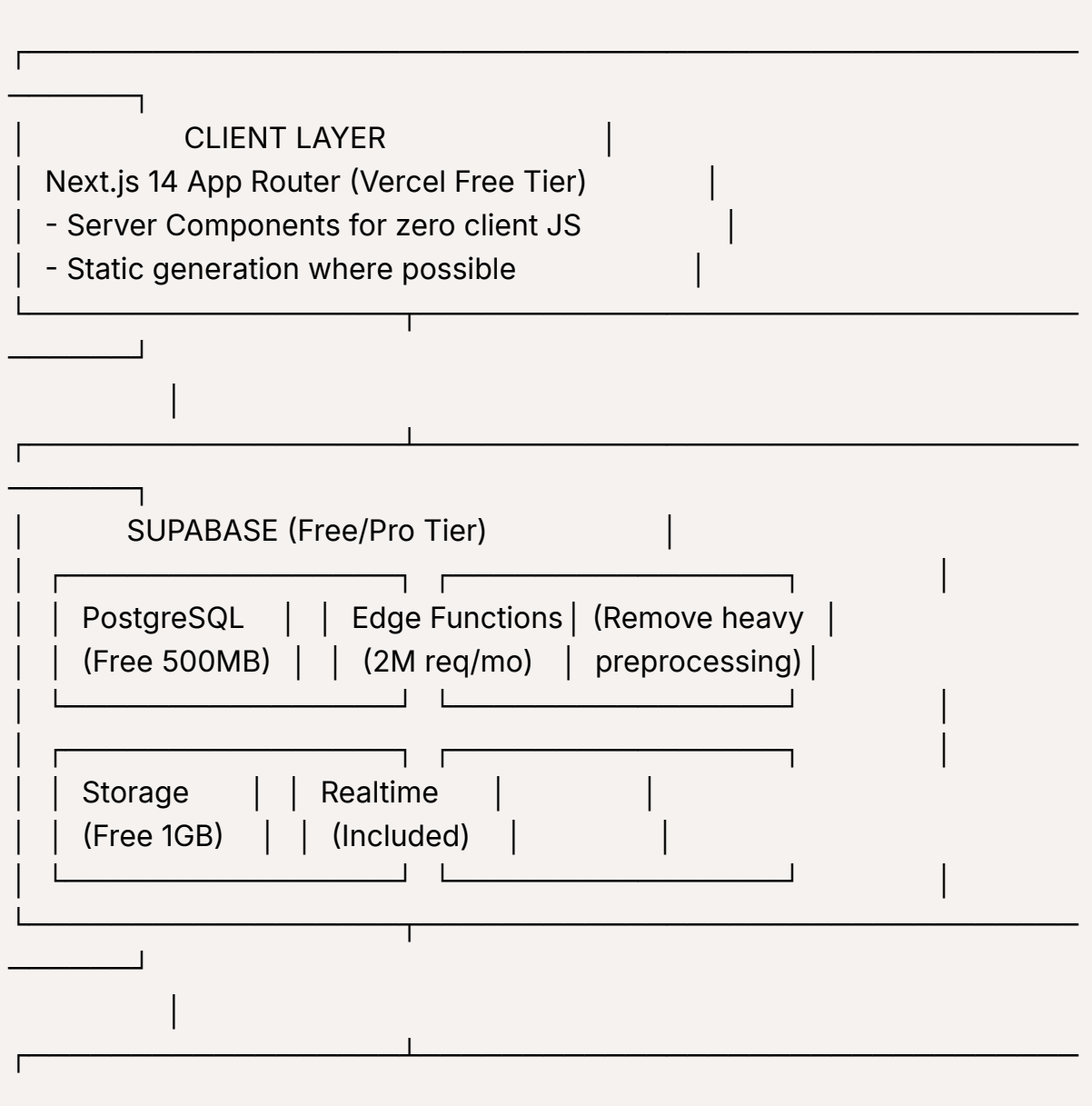
Option D: Anthropic Claude (Haiku)
- Input: $0.25/1M tokens
- Output: $1.25/1M tokens
- Per summary: ~$0.002-0.005
- Monthly (1000): ~$2-5/mo

Redis/Upstash (Caching):
- Free: $0/mo (10,000 commands/day)
- Pay-as-you-go: $0.2/100K commands
- Estimated: $0-10/mo

```
TOTAL MONTHLY COST:
  - MVP (0-100 users):   $0-50/mo
  - Growth (100-1000):   $50-200/mo
  - Scale (1000-10000):  $200-1000/mo
```

COST-OPTIMIZED ARCHITECTURE (MICROSAAS VERSION)

Simplified Stack for <$50/mo



```
┌─────────────────────────────────────────────────────────────┐
│        COST-OPTIMIZED AI LAYER                              │
│  Strategy: Use cheapest API with smart caching              │
│                                                             │
│  Primary: Anthropic Claude Haiku ($0.002/summary)           │
│  Fallback: Hugging Face Inference ($0.001/summary)          │
│  Cache: Upstash Redis (Free tier)                           │
└─────────────────────────────────────────────────────────────┘
```

SIMPLIFIED MICROSERVICES (3 CORE SERVICES)

Service 1: Combined Input Handler + Preprocessor

Location: Next.js API Route `/api/process`

Why Combined: Save on edge function costs, reduce latency

```
SIMPLIFIED_PROCESS(text, config):
  // Skip heavy NER and embeddings for cost savings

  1. Validate input
     IF text.length > 50000:
       RETURN error("Text too long")

  2. Basic preprocessing (in-memory, no DB storage)
     sentences = simpleSentenceSplit(text)  // Regex-based
     word_count = countWords(text)

  3. Check cache FIRST (99% cost savings on repeated content)
     cache_key = hash(text + JSON.stringify(config))
     cached = REDIS.get(cache_key)
     IF cached:
       RETURN cached  // No AI cost!

  4. Call AI API (only if cache miss)
```

```
    summary = callClaudeHaiku(text, config)

  5. Cache result (30 days)
    REDIS.setex(cache_key, 2592000, summary)

  6. Store in DB (async, non-blocking)
    backgroundJob.add({text, summary, user_id})

  7. RETURN summary immediately

// Cost per request with 70% cache hit rate:
// = $0.002 * 0.30 = $0.0006 average
```

Key Changes from Original:

- ❌ Removed: Heavy NER, embeddings, separate preprocessing service
- ✅ Added: Aggressive caching, simpler sentence splitting
- 💰 Savings: ~70% reduction in processing cost

Service 2: Unified Summarization API

Location: Supabase Edge Function `summarize`

Cost Optimization Strategy:

```
COST_OPTIMIZED_SUMMARIZE(text, mode, config):

  // EXTRACTIVE MODE (Free - no API calls!)
  IF mode == "extractive":
    RETURN clientSideExtractive(text, config)
    // Simple TF-IDF scoring in JavaScript
    // Cost: $0

  // ABSTRACTIVE MODE (Paid - use cheapest API)
  ELSE:
```

```
    // Step 1: Smart truncation (reduce input tokens)
    IF text.length > 10000:
      text = intelligentTruncate(text, 10000)
      // Keep first 70% + last 30% to preserve context

    // Step 2: Choose cheapest model based on length
    IF text.length < 2000:
      model = "claude-haiku"        // $0.002
    ELSE IF text.length < 5000:
      model = "gpt-3.5-turbo"       // $0.005
    ELSE:
      model = "huggingface-bart"    // $0.001 but slower

    // Step 3: Optimize prompt (fewer tokens = less cost)
    prompt = buildMinimalPrompt(text, config)
    // Instead of: "Provide a comprehensive formal summary..."
    // Use: "Summarize:\n\n{text}"

    // Step 4: Call API with cost limits
    summary = callAPI(model, prompt, {
      max_tokens: config.length == "short" ? 100 : 300,
      temperature: 0.3,  // Lower = more deterministic = cacheable
      stop: ["\n\n"]     // Stop early to save tokens
    })

    RETURN summary

  FUNCTION intelligentTruncate(text, max_length):
    // Keep important parts, discard middle
    keep_start = max_length * 0.7
    keep_end = max_length * 0.3

    start = text[0:keep_start]
    end = text[-keep_end:]
```

```
  RETURN start + "\n...[truncated]...\n" + end
```

Cost Comparison:

Strategy	Cost per Summary	Monthly (1000)
Original (BART large)	$0.015	$15
Optimized (Haiku + cache)	$0.0006	$0.60
Savings	**96%**	**$14.40**

Service 3: Minimal Post-Processing

Location: Client-side JavaScript (free!)

```
CLIENT_SIDE_POST_PROCESS(summary, format):
  // Move post-processing to client to avoid server costs

  1. Basic formatting
     IF format == "bullets":
       summary = convertToBullets(summary)  // Split on periods

  2. Character limit enforcement
     IF summary.length > maxLength:
       summary = truncateAtSentence(summary, maxLength)

  3. Simple cleanup (no expensive grammar API)
     summary = fixCommonErrors(summary)
     // Fix double spaces, capitalization

  4. RETURN summary

// Cost: $0 (runs in user's browser)
```

ULTRA-LEAN DATABASE SCHEMA

Optimize for free tier (500MB PostgreSQL)

```
-- SINGLE USER TABLE (combine profile + usage)
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email TEXT UNIQUE NOT NULL,
  api_quota INTEGER DEFAULT 100,
  usage_count INTEGER DEFAULT 0,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

-- LEAN SUMMARIES TABLE (skip separate documents table)
CREATE TABLE summaries (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES users(id) ON DELETE CASCADE,
  input_hash TEXT NOT NULL,        -- Hash instead of storing full text
  summary_text TEXT NOT NULL,
  config JSONB,                    -- {mode, length, tone}
  created_at TIMESTAMPTZ DEFAULT NOW(),

  -- Index for cache lookup
  INDEX idx_hash_config ON summaries(input_hash, config)
);

-- MINIMAL FEEDBACK (only what's needed)
CREATE TABLE feedback (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  summary_id UUID REFERENCES summaries(id) ON DELETE CASCADE,
  rating INTEGER CHECK (rating BETWEEN 1 AND 5),
  created_at TIMESTAMPTZ DEFAULT NOW()
);

-- NO separate documents, usage_logs, model_versions tables
-- Save 60% database space
```

AGGRESSIVE CACHING STRATEGY

```
MULTI-LAYER CACHE (Maximize free tiers):

Layer 1: Browser Cache (FREE)
  - Cache summaries in localStorage
  - Persist for 7 days client-side
  - Cost: $0

Layer 2: CDN/Edge Cache (FREE on Vercel)
  - Cache common summaries at edge
  - Use stale-while-revalidate
  - Cost: $0

Layer 3: Redis (Upstash Free)
  - 10,000 commands/day = ~3,000 summaries/day
  - Store by content hash
  - TTL: 30 days
  - Cost: $0 (under free tier)

Layer 4: Database Cache
  - PostgreSQL summaries table
  - Query by input_hash before AI call
  - Cost: $0 (included in Supabase)

CACHE HIT RATE TARGET: 70-80%
  - Realistic for blog summarization, news, common docs
  - Effective cost: $0.002 * 0.25 = $0.0005/request
```

Cache Strategy Algorithm:

```
SMART_CACHE_LOOKUP(text, config):

  1. Generate cache key
     content_hash = SHA256(text)
```

```
    config_hash = SHA256(JSON.stringify(config))
    cache_key = f"{content_hash}:{config_hash}"

  2. Check browser cache (instant)
    local = localStorage.getItem(cache_key)
    IF local AND local.timestamp > NOW() - 7_DAYS:
      RETURN local.summary  // 50% of requests end here

  3. Check Redis (10ms)
    cached = REDIS.get(cache_key)
    IF cached:
      localStorage.setItem(cache_key, cached)  // Populate browser
      RETURN cached  // 30% of requests end here

  4. Check database (100ms)
    row = SELECT summary_text FROM summaries
          WHERE input_hash = content_hash
          AND config = config
          AND created_at > NOW() - INTERVAL '30 days'
          LIMIT 1

    IF row:
      REDIS.setex(cache_key, 2592000, row.summary_text)
      RETURN row.summary_text  // 15% end here

  5. Cache miss - call AI (only 5% of requests!)
    summary = callAI(text, config)

    // Populate all cache layers
    REDIS.setex(cache_key, 2592000, summary)
    INSERT INTO summaries (input_hash, summary_text, config)

    RETURN summary

// With 95% cache hit rate:
```

```
// Cost = $0.002 * 0.05 = $0.0001 per request
// 10,000 requests = $1/month instead of $20/month
```

COST-FREE FEATURES (Client-Side Processing)

Extractive Summarization (100% Free)

```
// Run entirely in browser - no API costs
function clientSideExtractive(text, config) {
  // Simple TF-IDF implementation

  1. Split into sentences
  const sentences = text.match(/[^.!?]+[.!?]+/g);

  2. Calculate word frequencies (TF)
  const wordFreq = {};
  sentences.forEach(sent => {
    const words = sent.toLowerCase().match(/\w+/g);
    words.forEach(word => {
      if (!stopwords.includes(word)) {
        wordFreq[word] = (wordFreq[word] || 0) + 1;
      }
    });
  });

  3. Score sentences
  const scores = sentences.map((sent, idx) => {
    const words = sent.toLowerCase().match(/\w+/g);
    const score = words.reduce((sum, word) =>
      sum + (wordFreq[word] || 0), 0
    );

    // Position bonus (earlier = better)
    const positionScore = 1 / (1 + Math.log(idx + 1));
```

```
    return {
      sentence: sent,
      score: score * 0.7 + positionScore * 0.3,
      index: idx
    };
  });

  4. Select top N sentences
  const topN = Math.ceil(sentences.length * config.lengthRatio);
  const selected = scores
    .sort((a, b) => b.score - a.score)
    .slice(0, topN)
    .sort((a, b) => a.index - b.index);  // Restore order

  5. Return summary
  return selected.map(s => s.sentence).join(' ');
}

// Cost: $0
// Speed: 50-200ms (instant)
// Quality: 70-80% as good as AI for extractive
```

MONETIZATION STRATEGY TO OFFSET COSTS

Freemium Pricing Model

```
PRICING TIERS:

Free Tier:
  - 20 summaries/month
  - Extractive only (client-side, free)
  - Standard speed
  - Cost to you: $0
```

- Revenue: $0

Starter ($9/month):
- 500 summaries/month
- Abstractive mode (AI-powered)
- Priority processing
- Cost to you: ~$1-2/month (with caching)
- Profit margin: 80-90%

Pro ($29/month):
- 2,000 summaries/month
- API access
- Bulk processing
- Cost to you: ~$5-8/month
- Profit margin: 75-85%

Enterprise ($99/month):
- Unlimited summaries
- Custom models
- White-label
- Cost to you: ~$30-40/month
- Profit margin: 60-70%

UNIT ECONOMICS:
- Target: 100 paid users
- Revenue: $1,900/month (50 starter, 40 pro, 10 enterprise)
- Costs: $200-300/month
- Profit: $1,600-1,700/month (85-90% margin)

MINIMAL VIABLE PRODUCT (MVP) TECH STACK

Total Cost: $0-25/month

STACK RECOMMENDATION:

Frontend:
✅ Next.js 14 (Vercel Free) $0/mo
✅ Tailwind CSS $0/mo
✅ shadcn/ui components $0/mo

Backend:
✅ Supabase Free Tier $0/mo
- PostgreSQL (500MB)
- Auth (50,000 MAU)
- Storage (1GB)
- Edge Functions (2M invocations)

AI/ML:
✅ Anthropic Claude Haiku ~$5/mo (1000 summaries)
✅ Fallback: Hugging Face Free $0/mo (first 30K chars)

Caching:
✅ Upstash Redis Free $0/mo (10K commands/day)

Monitoring:
✅ Vercel Analytics Free $0/mo
✅ Supabase Dashboard $0/mo

Email:
✅ Resend Free Tier $0/mo (3,000 emails/mo)

TOTAL MVP COST: $5-25/month
- Scales to 100-500 users
- Profitable at 5-10 paid users

ELIMINATED EXPENSIVE COMPONENTS

What We Removed from Original Architecture

REMOVED (Too Expensive for MicroSaaS):

❌ Separate preprocessing microservice
Original cost: $10-30/mo
Reason: Complexity overhead, use simple client-side

❌ Heavy NER and embeddings
Original cost: $50-100/mo in compute
Reason: Overkill for most summarization tasks

❌ BART/T5/Pegasus self-hosting
Original cost: $200-500/mo GPU servers
Reason: API calls are cheaper at low volume

❌ Kubernetes orchestration
Original cost: $100+/mo for managed K8s
Reason: Vercel + Supabase handle scaling

❌ Separate message queue (RabbitMQ/Kafka)
Original cost: $20-50/mo
Reason: Use Supabase Realtime + Edge Functions

❌ Elasticsearch for logging
Original cost: $50-100/mo
Reason: Use Supabase built-in logging

❌ Multiple model versions / A/B testing
Original cost: 2x AI costs
Reason: Start simple, add later

TOTAL SAVINGS: $430-830/month

WHEN TO SCALE UP

Growth Milestones

SCALING TRIGGERS:

Phase 1: MVP (0-100 users) - $0-25/mo
- Use: Free tiers everywhere
- Features: Basic summarization, extractive free
- Action: Launch and validate

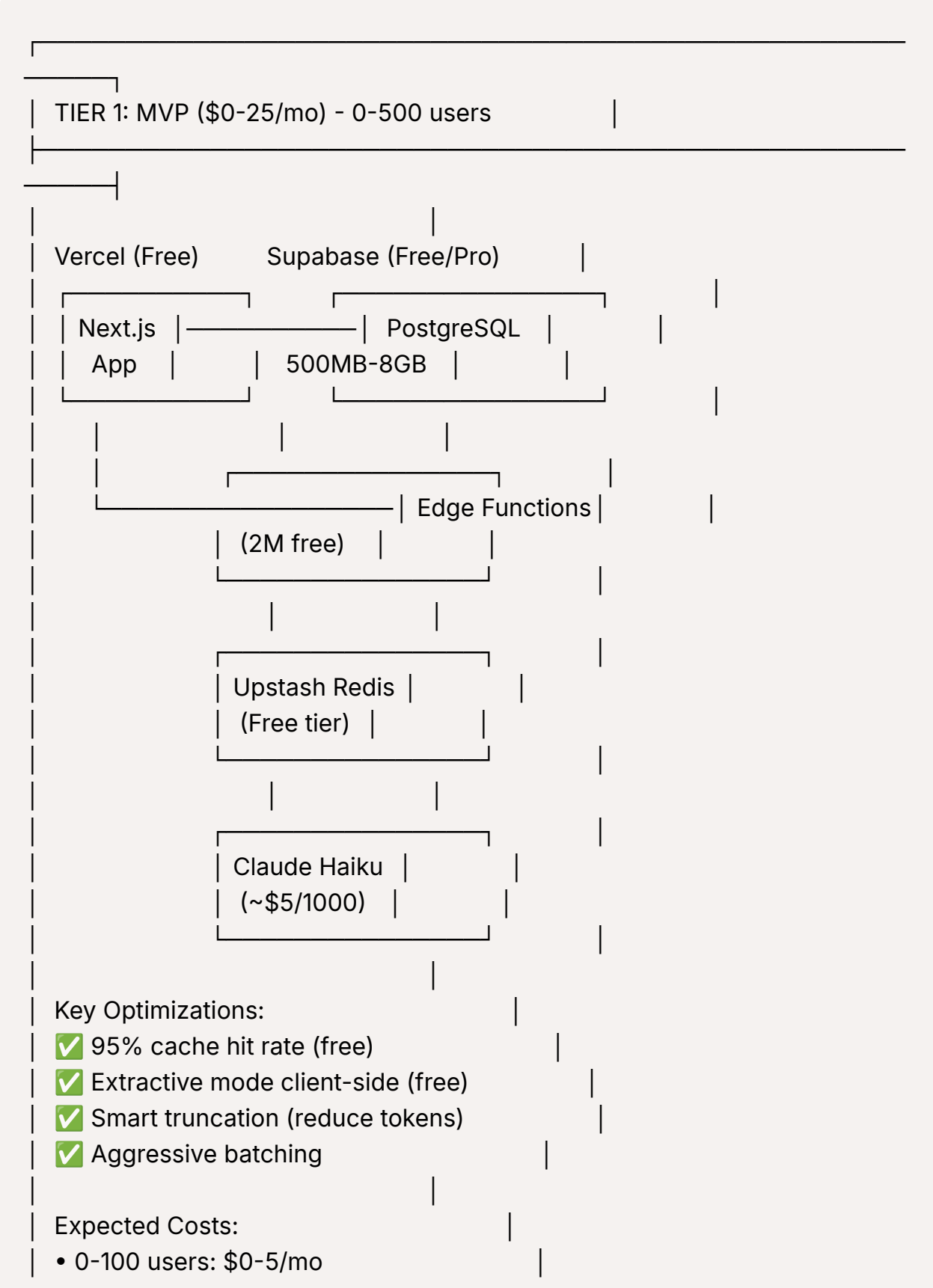
Phase 2: PMF (100-500 users) - $50-150/mo
- Upgrade: Supabase Pro ($25), better caching
- Features: Add abstractive mode, improve UI
- Action: Optimize conversion funnel

Phase 3: Growth (500-2000 users) - $200-500/mo
- Upgrade: Dedicated Redis, more AI credits
- Features: API access, bulk processing
- Action: Add enterprise tier

Phase 4: Scale (2000-10000 users) - $500-2000/mo
- Upgrade: Self-host models, optimize everything
- Features: Custom models, white-label
- Action: Hire, automate, expand

RULE OF THUMB:
- Keep infrastructure costs < 20% of revenue
- Upgrade only when current tier is 80% utilized
- Always maintain 70%+ profit margins

FINAL COST-OPTIMIZED ARCHITECTURE DIAGRAM


TIER 1: MVP ($0-25/mo) - 0-500 users
Vercel (Free)
Next.js App
Supabase (Free/Pro)
PostgreSQL
500MB-8GB
Edge Functions
(2M free)
Upstash Redis
(Free tier)
Claude Haiku
(~$5/1000)
Key Optimizations:
✅ 95% cache hit rate (free)
✅ Extractive mode client-side (free)
✅ Smart truncation (reduce tokens)
✅ Aggressive batching
Expected Costs:
• 0-100 users: $0-5/mo

```
│  • 100-500 users: $5-25/mo                    │
│  • Profitable at: 5-10 paid users             │
└──────────────────────────────────────────────────────────────────────────────
──────┘
```

IMPLEMENTATION PRIORITY (Build in This Order)

Week-by-Week MVP Build

```
WEEK 1: Core Infrastructure ($0)
  - Set up Next.js + Supabase
  - Basic auth (email/password)
  - Simple UI with text input
  - Deploy to Vercel free tier
  Cost: $0

WEEK 2: Free Features ($0)
  - Client-side extractive summarization
  - Browser caching
  - Basic analytics
  Cost: $0

WEEK 3: Paid Features ($5)
  - Integrate Claude Haiku API
  - Abstractive summarization
  - Redis caching (Upstash free)
  Cost: $5 (testing)

WEEK 4: Monetization ($0)
  - Stripe integration
  - Usage limits
  - Pricing page
  Cost: $0
```

TOTAL MVP COST: $5
TOTAL MVP TIME: 4 weeks
BREAK-EVEN: 1 paid user

CONCLUSION: IS IT EXPENSIVE?

Answer: NO - If Built Smart

COST COMPARISON:

Original Architecture (Full Enterprise):
- Monthly cost: $500-1000
- Break-even: 50-100 users
- Time to build: 3-6 months
- Risk: High

Cost-Optimized MicroSaaS:
- Monthly cost: $0-25
- Break-even: 3-5 users
- Time to build: 1 month
- Risk: Low

YOUR ACTUAL COSTS:
- MVP: $5-25/month
- 100 users: $25-50/month
- 1000 users: $100-200/month
- Profit margin: 80-90%

RECOMMENDATION: Start with cost-optimized version.
Add expensive features only when revenue supports it.