Contents

SECTION 32: TIME MACHINE CORE - DATABASE & SERVICE LAYER (v4.0.0)

Version: 4.0.0 | Apple Time Machine-inspired chat history for Think Tank
NEVER lose a chat or file - everything is preserved and recoverable forever

---

32.1 Time Machine Design Philosophy

Inspired by Apple Time Machine's Best Parts

```
                    APPLE TIME MACHINE INSPIRATION


 WHAT WE'RE BORROWING:

 1. Visual "fly back through time" - messages recede into the past
 2. Calendar-based navigation - pick any date to jump to
 3. Timeline bar on the side - scrub to any point
 4. One-click restore - instantly recover anything
 5. "Enter Time Machine" mode - separate from normal view
 6. Everything is automatic - no manual "save" needed
 7. Never delete anything - space is cheap, data is priceless

 RADIANT IMPROVEMENTS:

 1. Works for chat AND files (unified versioning)
 2. API-first - client apps can build their own Time Machine UI
```

```
3. AI-aware - simplified API lets AI help users navigate history
4. Real-time - see changes as they happen, not just hourly backups
5. Granular - restore single message OR entire conversation
6. Searchable - find that thing you said 3 months ago
```

The Golden Rules

1. **AUTOMATIC** - Every action creates a snapshot. Users never "save."
2. **INVISIBLE** - Hidden until needed. Default UI is just simple chat.
3. **COMPLETE** - Messages, files, edits, metadata - everything versioned.
4. **INSTANT** - Restore happens in milliseconds, not minutes.
5. **FOREVER** - Nothing is ever truly deleted. Soft-delete only.

32.2 Core Types

```typescript
// packages/shared/src/types/time-machine.ts

//
// TIME MACHINE CORE TYPES
//

export type SnapshotTrigger =
  | 'message_sent'       // User sent a message
  | 'message_received'   // AI responded
  | 'message_edited'     // User edited a message
  | 'message_deleted'    // User "deleted" (soft) a message
  | 'file_uploaded'      // User uploaded a file
  | 'file_generated'     // AI generated a file
  | 'file_deleted'       // User "deleted" (soft) a file
  | 'chat_renamed'       // Chat title changed
  | 'restore_performed'  // User restored from history
  | 'manual_snapshot';   // User explicitly saved a point

export type RestoreScope =
  | 'full_chat'          // Restore entire chat to that point
  | 'single_message'     // Restore just one message
  | 'single_file'        // Restore just one file
  | 'message_range'      // Restore a range of messages
  | 'files_only';        // Restore all files, keep messages

export type MediaStatus =
  | 'active'             // Currently visible to user
  | 'processing'         // Being uploaded/processed
  | 'archived'           // Moved to cold storage (still retrievable)
  | 'soft_deleted';      // User "deleted" but still exists
```

```
export type ExportFormat = 'zip' | 'json' | 'markdown' | 'pdf' | 'html';

//
// SNAPSHOT - Point in time capture of chat state
//

export interface TimeMachineSnapshot {
  id: string;
  chatId: string;
  tenantId: string;

  // Version info
  version: number;                    // Monotonically increasing
  timestamp: string;                  // ISO 8601 with milliseconds

  // State summary at this point
  messageCount: number;
  fileCount: number;
  totalTokens: number;

  // What triggered this snapshot
  trigger: SnapshotTrigger;
  triggerDetails?: {
    messageId?: string;
    fileId?: string;
    description?: string;
  };

  // Lineage
  previousSnapshotId?: string;
  restoredFromSnapshotId?: string;    // If this was created by a restore

  // Integrity
  checksum: string;                   // SHA-256 of content

  // Metadata
  createdAt: string;
}

//
// MESSAGE VERSION - Every edit creates a new version
//

export interface MessageVersion {
  id: string;
  messageId: string;                  // Stable ID across versions
  tenantId: string;
  snapshotId: string;

  // Content
  content: string;
  role: 'user' | 'assistant' | 'system';
  modelId?: string;

  // Version info
  version: number;
  isActive: boolean;                  // Is this the current version?
  isSoftDeleted: boolean;

  // Edit tracking
  editReason?: string;
  editedBy?: string;

  // Timestamps
  createdAt: string;
  supersededAt?: string;
}

//
// MEDIA VAULT - Every file version preserved forever
//

export interface MediaVaultFile {
  id: string;
  chatId: string;
  tenantId: string;
  messageId?: string;
  snapshotId: string;

  // File identity
  originalName: string;               // What user named it
  displayName: string;                // What's shown in UI

  // S3 storage with versioning
  s3Bucket: string;
  s3Key: string;
  s3VersionId: string;                // Critical for immutability

  // File properties
  mimeType: string;
  sizeBytes: number;
  checksumSha256: string;

  // Preview
  thumbnailS3Key?: string;
  previewGenerated: boolean;

  // Version info
  version: number;
  previousVersionId?: string;

  // Source
  source: 'user_upload' | 'ai_generated' | 'system';

  // Status
  status: MediaStatus;

  // AI-enhanced metadata
  extractedText?: string;             // For searchability
  aiDescription?: string;             // AI-generated description

  // Timestamps
  createdAt: string;
  archivedAt?: string;
}

//
// TIMELINE - Complete history of a chat
//

export interface ChatTimeline {
  chatId: string;
  chatTitle: string;

  // Current state
  currentVersion: number;
  currentMessageCount: number;
  currentFileCount: number;

  // History
  snapshots: TimeMachineSnapshot[];

  // Aggregates
  totalSnapshots: number;
  totalMediaBytes: number;
  oldestSnapshot: string;             // ISO timestamp
  newestSnapshot: string;

  // Calendar data for navigation
  snapshotsByDate: Record<string, number>;  // "2024-12-23" -> count
}

//
```

```
// RESTORE REQUEST/RESULT
//

export interface RestoreRequest {
  chatId: string;
  targetSnapshotId: string;
  scope: RestoreScope;

  // For partial restores
  messageIds?: string[];
  fileIds?: string[];

  // Reason tracking
  reason?: string;
}

export interface RestoreResult {
  success: boolean;
  newSnapshotId: string;

  // What was restored
  messagesRestored: number;
  filesRestored: number;

  // The new current state
  newVersion: number;

  // For undo
  previousSnapshotId: string;
}

//
// EXPORT BUNDLE
//

export interface ExportBundle {
  id: string;
  chatId: string;
  tenantId: string;
  userId: string;

  // Scope
  fromSnapshotId?: string;            // null = from beginning
  toSnapshotId: string;

  // Format
  format: ExportFormat;
  includeMedia: boolean;
  includeVersionHistory: boolean;

  // File
  s3Key: string;
  sizeBytes: number;
  downloadCount: number;

  // Expiry
  expiresAt: string;

  // Status
  status: 'pending' | 'processing' | 'ready' | 'expired' | 'failed';
  errorMessage?: string;

  // Timestamps
  createdAt: string;
  completedAt?: string;
}
```

32.3 Database Schema

```
-- Migration 013: Time Machine for Think Tank
--

--
-- CHAT SNAPSHOTS - Point-in-time state captures (like Time Machine backups)
--

CREATE TABLE tm_snapshots (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id UUID NOT NULL REFERENCES tenants(id),
  chat_id UUID NOT NULL REFERENCES thinktank_chats(id) ON DELETE CASCADE,

  -- Version info
  version INTEGER NOT NULL,
  snapshot_timestamp TIMESTAMPTZ NOT NULL DEFAULT NOW(),

  -- State summary
  message_count INTEGER NOT NULL DEFAULT 0,
  file_count INTEGER NOT NULL DEFAULT 0,
  total_tokens BIGINT NOT NULL DEFAULT 0,

  -- Trigger
  trigger TEXT NOT NULL CHECK (trigger IN (
    'message_sent', 'message_received', 'message_edited', 'message_deleted',
    'file_uploaded', 'file_generated', 'file_deleted', 'chat_renamed',
    'restore_performed', 'manual_snapshot'
  )),
  trigger_message_id UUID,
  trigger_file_id UUID,
  trigger_description TEXT,

  -- Lineage
  previous_snapshot_id UUID REFERENCES tm_snapshots(id),
  restored_from_snapshot_id UUID REFERENCES tm_snapshots(id),

  -- Integrity
  checksum TEXT NOT NULL,

  -- Timestamps
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

  -- Constraints
  UNIQUE(chat_id, version)
);

-- Indexes for Time Machine navigation
CREATE INDEX idx_tm_snapshots_chat_version ON tm_snapshots(chat_id, version DESC);
CREATE INDEX idx_tm_snapshots_chat_timestamp ON tm_snapshots(chat_id, snapshot_timestamp DESC)
CREATE INDEX idx_tm_snapshots_date ON tm_snapshots(DATE(snapshot_timestamp), chat_id);

--
-- MESSAGE VERSIONS - Every edit preserved
--

CREATE TABLE tm_message_versions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id UUID NOT NULL REFERENCES tenants(id),
  message_id UUID NOT NULL,           -- Stable ID across versions
  snapshot_id UUID NOT NULL REFERENCES tm_snapshots(id) ON DELETE CASCADE,

  -- Content
  content TEXT NOT NULL,
  role TEXT NOT NULL CHECK (role IN ('user', 'assistant', 'system')),
  model_id TEXT,

  -- Metadata
  metadata JSONB DEFAULT '{}',

  -- Version info
  version INTEGER NOT NULL,
  is_active BOOLEAN NOT NULL DEFAULT TRUE,
  is_soft_deleted BOOLEAN NOT NULL DEFAULT FALSE,
  -- Edit tracking
  edit_reason TEXT,
  edited_by UUID REFERENCES users(id),

  -- Timestamps
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  superseded_at TIMESTAMPTZ,
  original_created_at TIMESTAMPTZ NOT NULL,  -- When message was first created

  -- Constraints
  UNIQUE(message_id, version)
);

-- Indexes for message lookup
CREATE INDEX idx_tm_messages_message_id ON tm_message_versions(message_id, version DESC);
CREATE INDEX idx_tm_messages_snapshot ON tm_message_versions(snapshot_id);
CREATE INDEX idx_tm_messages_active ON tm_message_versions(message_id) WHERE is_active = TRUE;
CREATE INDEX idx_tm_messages_search ON tm_message_versions USING gin(to_tsvector('english', co

--
-- MEDIA VAULT - Every file version preserved forever
--

CREATE TABLE tm_media_vault (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id UUID NOT NULL REFERENCES tenants(id),
  chat_id UUID NOT NULL REFERENCES thinktank_chats(id) ON DELETE CASCADE,
  message_id UUID,                    -- Can be NULL for chat-level files
  snapshot_id UUID NOT NULL REFERENCES tm_snapshots(id) ON DELETE CASCADE,

  -- File identity
  original_name TEXT NOT NULL,
  display_name TEXT NOT NULL,

  -- S3 storage (with versioning enabled on bucket)
  s3_bucket TEXT NOT NULL,
  s3_key TEXT NOT NULL,
  s3_version_id TEXT NOT NULL,        -- S3 object version for immutability

  -- File properties
  mime_type TEXT NOT NULL,
  size_bytes BIGINT NOT NULL,
  checksum_sha256 TEXT NOT NULL,

  -- Preview
  thumbnail_s3_key TEXT,
  preview_generated BOOLEAN DEFAULT FALSE,
  -- Version info
  version INTEGER NOT NULL,
  previous_version_id UUID REFERENCES tm_media_vault(id),

  -- Source
  source TEXT NOT NULL CHECK (source IN ('user_upload', 'ai_generated', 'system')),

  -- Status
  status TEXT NOT NULL DEFAULT 'active' CHECK (status IN (
    'active', 'processing', 'archived', 'soft_deleted'
  )),

  -- AI-enhanced metadata
  extracted_text TEXT,
  ai_description TEXT,
  metadata JSONB DEFAULT '{}',

  -- Timestamps
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  archived_at TIMESTAMPTZ,

  -- Constraints
  UNIQUE(chat_id, original_name, version)
);

-- Indexes for media lookup
CREATE INDEX idx_tm_media_chat ON tm_media_vault(chat_id);
CREATE INDEX idx_tm_media_snapshot ON tm_media_vault(snapshot_id);
CREATE INDEX idx_tm_media_name ON tm_media_vault(chat_id, original_name, version DESC);
CREATE INDEX idx_tm_media_search ON tm_media_vault USING gin(
  to_tsvector('english', COALESCE(extracted_text, '') || ' ' || COALESCE(ai_description, ''))
);

--
-- MESSAGE-MEDIA REFERENCES - Links messages to specific file versions
--

CREATE TABLE tm_message_media_refs (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  message_version_id UUID NOT NULL REFERENCES tm_message_versions(id) ON DELETE CASCADE,
  media_vault_id UUID NOT NULL REFERENCES tm_media_vault(id) ON DELETE CASCADE,

  -- Display order and type
  display_order INTEGER NOT NULL DEFAULT 0,
  reference_type TEXT NOT NULL CHECK (reference_type IN (
    'attachment',      -- User attached this file
    'inline',          -- Embedded in message content
    'result',          -- AI-generated result
    'reference'       -- Referenced but not attached
  )),

  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

  UNIQUE(message_version_id, media_vault_id)
);

--
-- RESTORE LOG - Audit trail for all restores
--

CREATE TABLE tm_restore_log (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id UUID NOT NULL REFERENCES tenants(id),
  chat_id UUID NOT NULL REFERENCES thinktank_chats(id) ON DELETE CASCADE,
  user_id UUID NOT NULL REFERENCES users(id),

  -- What was restored
  from_snapshot_id UUID NOT NULL REFERENCES tm_snapshots(id),
  to_snapshot_id UUID NOT NULL REFERENCES tm_snapshots(id),

  -- Scope
  scope TEXT NOT NULL CHECK (scope IN (
    'full_chat', 'single_message', 'single_file', 'message_range', 'files_only'
  )),

  -- Items restored
  message_ids UUID[],
  file_ids UUID[],
  messages_restored INTEGER DEFAULT 0,
  files_restored INTEGER DEFAULT 0,

  -- Reason
  reason TEXT,

  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

--
-- EXPORT BUNDLES - Track export requests
--

CREATE TABLE tm_export_bundles (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id UUID NOT NULL REFERENCES tenants(id),
  chat_id UUID NOT NULL REFERENCES thinktank_chats(id) ON DELETE CASCADE,
  user_id UUID NOT NULL REFERENCES users(id),
```

```sql
  -- Scope
  from_snapshot_id UUID REFERENCES tm_snapshots(id),
  to_snapshot_id UUID NOT NULL REFERENCES tm_snapshots(id),

  -- Format
  format TEXT NOT NULL CHECK (format IN ('zip', 'json', 'markdown', 'pdf', 'html')),
  include_media BOOLEAN DEFAULT TRUE,
  include_version_history BOOLEAN DEFAULT FALSE,

  -- File
  s3_key TEXT,
  size_bytes BIGINT DEFAULT 0,
  download_count INTEGER DEFAULT 0,

  -- Status
  status TEXT NOT NULL DEFAULT 'pending' CHECK (status IN (
    'pending', 'processing', 'ready', 'expired', 'failed'
  )),
  error_message TEXT,

  -- Expiry
  expires_at TIMESTAMPTZ NOT NULL DEFAULT (NOW() + INTERVAL '7 days'),

  -- Timestamps
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  completed_at TIMESTAMPTZ
);

--
-- ROW LEVEL SECURITY
--

ALTER TABLE tm_snapshots ENABLE ROW LEVEL SECURITY;
ALTER TABLE tm_message_versions ENABLE ROW LEVEL SECURITY;
ALTER TABLE tm_media_vault ENABLE ROW LEVEL SECURITY;
ALTER TABLE tm_message_media_refs ENABLE ROW LEVEL SECURITY;
ALTER TABLE tm_restore_log ENABLE ROW LEVEL SECURITY;
ALTER TABLE tm_export_bundles ENABLE ROW LEVEL SECURITY;

-- Tenant isolation policies
CREATE POLICY tm_snapshots_tenant ON tm_snapshots
  USING (tenant_id = current_setting('app.current_tenant_id')::UUID);

CREATE POLICY tm_message_versions_tenant ON tm_message_versions
  USING (tenant_id = current_setting('app.current_tenant_id')::UUID);

CREATE POLICY tm_media_vault_tenant ON tm_media_vault
```

```sql
  USING (tenant_id = current_setting('app.current_tenant_id')::UUID);

CREATE POLICY tm_message_media_refs_tenant ON tm_message_media_refs
  USING (EXISTS (
    SELECT 1 FROM tm_message_versions mv
    WHERE mv.id = tm_message_media_refs.message_version_id
    AND mv.tenant_id = current_setting('app.current_tenant_id')::UUID
  ));

CREATE POLICY tm_restore_log_tenant ON tm_restore_log
  USING (tenant_id = current_setting('app.current_tenant_id')::UUID);

CREATE POLICY tm_export_bundles_tenant ON tm_export_bundles
  USING (tenant_id = current_setting('app.current_tenant_id')::UUID);

--
-- VIEWS FOR COMMON QUERIES
--

-- Current state view (what users see in normal mode)
CREATE VIEW tm_current_messages AS
SELECT
  mv.message_id,
  mv.content,
  mv.role,
  mv.model_id,
  mv.metadata,
  mv.version,
  mv.original_created_at,
  mv.created_at as version_created_at,
  s.chat_id,
  s.tenant_id
FROM tm_message_versions mv
JOIN tm_snapshots s ON mv.snapshot_id = s.id
WHERE mv.is_active = TRUE AND mv.is_soft_deleted = FALSE;

-- Files with version count
CREATE VIEW tm_files_with_versions AS
SELECT
  mv.*,
  (SELECT COUNT(*) FROM tm_media_vault
   WHERE chat_id = mv.chat_id AND original_name = mv.original_name) as version_count
FROM tm_media_vault mv
WHERE mv.status = 'active';

-- Calendar view for timeline navigation
CREATE VIEW tm_calendar_view AS
SELECT
```

```sql
  chat_id,
  DATE(snapshot_timestamp) as snapshot_date,
  COUNT(*) as snapshot_count,
  MIN(snapshot_timestamp) as first_snapshot,
  MAX(snapshot_timestamp) as last_snapshot
FROM tm_snapshots
GROUP BY chat_id, DATE(snapshot_timestamp)
ORDER BY snapshot_date DESC;
```

32.4 Time Machine Service (Core Business Logic)

```typescript
// packages/functions/src/services/time-machine.service.ts

import { Pool, PoolClient } from 'pg';
import { S3Client, PutObjectCommand, GetObjectCommand, CopyObjectCommand, HeadObjectCommand }
import { getSignedUrl } from '@aws-sdk/s3-request-presigner';
import { createHash } from 'crypto';
import { v4 as uuid } from 'uuid';
import {
  TimeMachineSnapshot,
  MessageVersion,
  MediaVaultFile,
  ChatTimeline,
  RestoreRequest,
  RestoreResult,
  ExportBundle,
  SnapshotTrigger,
  RestoreScope,
  ExportFormat,
} from '@radiant/shared';

//
// TIME MACHINE SERVICE
//

export class TimeMachineService {
  private pool: Pool;
  private s3: S3Client;
  private bucketName: string;

  constructor(pool: Pool) {
    this.pool = pool;
    this.s3 = new S3Client({});
    this.bucketName = process.env.MEDIA_VAULT_BUCKET!;
  }
```

```
//
// SNAPSHOT CREATION (Automatic on every action)
//

async createSnapshot(params: {
  chatId: string;
  tenantId: string;
  trigger: SnapshotTrigger;
  triggerMessageId?: string;
  triggerFileId?: string;
  triggerDescription?: string;
}): Promise<TimeMachineSnapshot> {
  const client = await this.pool.connect();

  try {
    await client.query('BEGIN');
    await client.query(`SET app.current_tenant_id = '${params.tenantId}'`);

    // Get previous snapshot
    const prevResult = await client.query(`
      SELECT id, version FROM tm_snapshots
      WHERE chat_id = $1
      ORDER BY version DESC LIMIT 1
    `, [params.chatId]);

    const prevSnapshot = prevResult.rows[0];
    const newVersion = prevSnapshot ? prevSnapshot.version + 1 : 1;

    // Count current state
    const countsResult = await client.query(`
      SELECT
        (SELECT COUNT(DISTINCT message_id) FROM tm_message_versions mv
         JOIN tm_snapshots s ON mv.snapshot_id = s.id
         WHERE s.chat_id = $1 AND mv.is_active = TRUE AND mv.is_soft_deleted = FALSE) as mess
        (SELECT COUNT(*) FROM tm_media_vault
         WHERE chat_id = $1 AND status = 'active') as file_count,
        (SELECT COALESCE(SUM((metadata->>'tokens')::bigint), 0) FROM tm_message_versions mv
         JOIN tm_snapshots s ON mv.snapshot_id = s.id
         WHERE s.chat_id = $1 AND mv.is_active = TRUE) as total_tokens
    `, [params.chatId]);

    const counts = countsResult.rows[0];

    // Compute checksum of current state
    const checksum = await this.computeChatChecksum(client, params.chatId);

    // Create snapshot
    const result = await client.query(`
      INSERT INTO tm_snapshots (
        tenant_id, chat_id, version, message_count, file_count, total_tokens,
        trigger, trigger_message_id, trigger_file_id, trigger_description,
        previous_snapshot_id, checksum
      ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12)
      RETURNING *
    `, [
      params.tenantId,
      params.chatId,
      newVersion,
      parseInt(counts.message_count) || 0,
      parseInt(counts.file_count) || 0,
      parseInt(counts.total_tokens) || 0,
      params.trigger,
      params.triggerMessageId,
      params.triggerFileId,
      params.triggerDescription,
      prevSnapshot?.id,
      checksum,
    ]);

    await client.query('COMMIT');

    return this.mapSnapshotRow(result.rows[0]);
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

private async computeChatChecksum(client: PoolClient, chatId: string): Promise<string> {
  const result = await client.query(`
    SELECT mv.message_id, mv.content, mv.role, mv.version
    FROM tm_message_versions mv
    JOIN tm_snapshots s ON mv.snapshot_id = s.id
    WHERE s.chat_id = $1 AND mv.is_active = TRUE AND mv.is_soft_deleted = FALSE
    ORDER BY mv.original_created_at ASC
  `, [chatId]);

  const hash = createHash('sha256');
  for (const row of result.rows) {
    hash.update(`${row.message_id}:${row.content}:${row.role}:${row.version}|`);
  }
  return hash.digest('hex');
}
```

```
//
// MESSAGE VERSIONING
//

async saveMessageVersion(params: {
  chatId: string;
  tenantId: string;
  messageId: string;
  content: string;
  role: 'user' | 'assistant' | 'system';
  modelId?: string;
  metadata?: Record<string, unknown>;
  isEdit?: boolean;
  editReason?: string;
  editedBy?: string;
}): Promise<MessageVersion> {
  const client = await this.pool.connect();

  try {
    await client.query('BEGIN');
    await client.query(`SET app.current_tenant_id = '${params.tenantId}'`);

    // Get or create snapshot
    let snapshot = await this.getLatestSnapshot(client, params.chatId);
    if (!snapshot) {
      // Create initial snapshot
      await client.query('COMMIT');
      snapshot = await this.createSnapshot({
        chatId: params.chatId,
        tenantId: params.tenantId,
        trigger: params.role === 'user' ? 'message_sent' : 'message_received',
      });
      await client.query('BEGIN');
      await client.query(`SET app.current_tenant_id = '${params.tenantId}'`);
    }

    // Get previous version of this message (if editing)
    const prevResult = await client.query(`
      SELECT id, version FROM tm_message_versions
      WHERE message_id = $1 AND is_active = TRUE
      ORDER BY version DESC LIMIT 1
    `, [params.messageId]);

    const prevVersion = prevResult.rows[0];
    const newVersion = prevVersion ? prevVersion.version + 1 : 1;

    // If editing, mark previous as superseded
    if (prevVersion) {
      await client.query(`
        UPDATE tm_message_versions
        SET is_active = FALSE, superseded_at = NOW()
        WHERE id = $1
      `, [prevVersion.id]);
    }

    // Insert new version
    const result = await client.query(`
      INSERT INTO tm_message_versions (
        tenant_id, message_id, snapshot_id, content, role, model_id,
        metadata, version, is_active, edit_reason, edited_by, original_created_at
      ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, TRUE, $9, $10,
        COALESCE((SELECT original_created_at FROM tm_message_versions WHERE message_id = $2 
      )
      RETURNING *
    `, [
      params.tenantId,
      params.messageId,
      snapshot.id,
      params.content,
      params.role,
      params.modelId,
      JSON.stringify(params.metadata || {}),
      newVersion,
      params.editReason,
      params.editedBy,
    ]);

    await client.query('COMMIT');

    // Create snapshot for this change
    await this.createSnapshot({
      chatId: params.chatId,
      tenantId: params.tenantId,
      trigger: params.isEdit ? 'message_edited' : (params.role === 'user' ? 'message_sent' :
      triggerMessageId: params.messageId,
    });

    return this.mapMessageVersionRow(result.rows[0]);
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}
```

```
async softDeleteMessage(params: {
  chatId: string;
  tenantId: string;
  messageId: string;
  deletedBy: string;
}): Promise<void> {
  const client = await this.pool.connect();

  try {
    await client.query('BEGIN');
    await client.query(`SET app.current_tenant_id = '${params.tenantId}'`);

    // Mark as soft deleted (NEVER actually delete)
    await client.query(`
      UPDATE tm_message_versions
      SET is_soft_deleted = TRUE, superseded_at = NOW()
      WHERE message_id = $1 AND is_active = TRUE
    `, [params.messageId]);

    await client.query('COMMIT');

    // Create snapshot
    await this.createSnapshot({
      chatId: params.chatId,
      tenantId: params.tenantId,
      trigger: 'message_deleted',
      triggerMessageId: params.messageId,
    });
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

//
// MEDIA VAULT
//

async uploadFile(params: {
  chatId: string;
  tenantId: string;
  messageId?: string;
  file: {
    name: string;
    data: Buffer;
    mimeType: string;
  };
  source: 'user_upload' | 'ai_generated' | 'system';
}): Promise<MediaVaultFile> {
  const client = await this.pool.connect();

  try {
    await client.query('BEGIN');
    await client.query(`SET app.current_tenant_id = '${params.tenantId}'`);

    // Get or create snapshot
    let snapshot = await this.getLatestSnapshot(client, params.chatId);
    if (!snapshot) {
      await client.query('COMMIT');
      snapshot = await this.createSnapshot({
        chatId: params.chatId,
        tenantId: params.tenantId,
        trigger: 'file_uploaded',
      });
      await client.query('BEGIN');
      await client.query(`SET app.current_tenant_id = '${params.tenantId}'`);
    }

    // Check for existing versions
    const existingResult = await client.query(`
      SELECT id, version FROM tm_media_vault
      WHERE chat_id = $1 AND original_name = $2
      ORDER BY version DESC LIMIT 1
    `, [params.chatId, params.file.name]);

    const existing = existingResult.rows[0];
    const newVersion = existing ? existing.version + 1 : 1;

    // Compute checksum
    const checksum = createHash('sha256').update(params.file.data).digest('hex');

    // Generate S3 key
    const fileId = uuid();
    const s3Key = `${params.tenantId}/${params.chatId}/${fileId}/${params.file.name}`;

    // Upload to S3 (bucket has versioning enabled)
    const putResult = await this.s3.send(new PutObjectCommand({
      Bucket: this.bucketName,
      Key: s3Key,
      Body: params.file.data,
      ContentType: params.file.mimeType,
      Metadata: {
        'chat-id': params.chatId,
        'original-name': params.file.name,
        'version': String(newVersion),
        'checksum': checksum,
      },
    }));

    // Insert into media vault
    const result = await client.query(`
      INSERT INTO tm_media_vault (
        tenant_id, chat_id, message_id, snapshot_id, original_name, display_name,
        s3_bucket, s3_key, s3_version_id, mime_type, size_bytes, checksum_sha256,
        version, previous_version_id, source
      ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14, $15)
      RETURNING *
    `, [
      params.tenantId,
      params.chatId,
      params.messageId,
      snapshot.id,
      params.file.name,
      params.file.name,
      this.bucketName,
      s3Key,
      putResult.VersionId!,
      params.file.mimeType,
      params.file.data.length,
      checksum,
      newVersion,
      existing?.id,
      params.source,
    ]);

    await client.query('COMMIT');

    // Create snapshot
    await this.createSnapshot({
      chatId: params.chatId,
      tenantId: params.tenantId,
      trigger: params.source === 'user_upload' ? 'file_uploaded' : 'file_generated',
      triggerFileId: result.rows[0].id,
    });

    return this.mapMediaVaultRow(result.rows[0]);
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
```

```
}

async getFileDownloadUrl(fileId: string, expiresIn = 3600): Promise<string> {
  const result = await this.pool.query(`
    SELECT s3_bucket, s3_key, s3_version_id FROM tm_media_vault WHERE id = $1
  `, [fileId]);

  if (!result.rows[0]) {
    throw new Error('File not found');
  }

  const { s3_bucket, s3_key, s3_version_id } = result.rows[0];

  const command = new GetObjectCommand({
    Bucket: s3_bucket,
    Key: s3_key,
    VersionId: s3_version_id,
  });

  return getSignedUrl(this.s3, command, { expiresIn });
}

async getFileVersions(chatId: string, fileName: string): Promise<MediaVaultFile[]> {
  const result = await this.pool.query(`
    SELECT * FROM tm_media_vault
    WHERE chat_id = $1 AND original_name = $2
    ORDER BY version DESC
  `, [chatId, fileName]);

  return result.rows.map(row => this.mapMediaVaultRow(row));
}

//
// TIMELINE NAVIGATION (The "fly back through time" experience)
//

async getTimeline(chatId: string, tenantId: string): Promise<ChatTimeline> {
  await this.pool.query(`SET app.current_tenant_id = '${tenantId}'`);

  // Get chat info
  const chatResult = await this.pool.query(`
    SELECT title FROM thinktank_chats WHERE id = $1
  `, [chatId]);

  // Get all snapshots
  const snapshotsResult = await this.pool.query(`
    SELECT * FROM tm_snapshots
    WHERE chat_id = $1
```

```
      ORDER BY version ASC
    `, [chatId]);

    // Get calendar data
    const calendarResult = await this.pool.query(`
      SELECT snapshot_date::text, snapshot_count
      FROM tm_calendar_view
      WHERE chat_id = $1
    `, [chatId]);

    // Get total media size
    const sizeResult = await this.pool.query(`
      SELECT COALESCE(SUM(size_bytes), 0) as total_size
      FROM tm_media_vault
      WHERE chat_id = $1
    `, [chatId]);

    const snapshots = snapshotsResult.rows.map(row => this.mapSnapshotRow(row));
    const currentSnapshot = snapshots[snapshots.length - 1];

    const snapshotsByDate: Record<string, number> = {};
    for (const row of calendarResult.rows) {
      snapshotsByDate[row.snapshot_date] = parseInt(row.snapshot_count);
    }

    return {
      chatId,
      chatTitle: chatResult.rows[0]?.title || 'Untitled Chat',
      currentVersion: currentSnapshot?.version || 0,
      currentMessageCount: currentSnapshot?.messageCount || 0,
      currentFileCount: currentSnapshot?.fileCount || 0,
      snapshots,
      totalSnapshots: snapshots.length,
      totalMediaBytes: parseInt(sizeResult.rows[0].total_size) || 0,
      oldestSnapshot: snapshots[0]?.timestamp || new Date().toISOString(),
      newestSnapshot: currentSnapshot?.timestamp || new Date().toISOString(),
      snapshotsByDate,
    };
  }

  async getChatAtSnapshot(chatId: string, snapshotId: string, tenantId: string): Promise<{
    snapshot: TimeMachineSnapshot;
    messages: MessageVersion[];
    files: MediaVaultFile[];
  }> {
    await this.pool.query(`SET app.current_tenant_id = '${tenantId}'`);

    // Get snapshot
    const snapshotResult = await this.pool.query(`
      SELECT * FROM tm_snapshots WHERE id = $1
    `, [snapshotId]);

    if (!snapshotResult.rows[0]) {
      throw new Error('Snapshot not found');
    }

    // Get messages at this snapshot
    // This requires understanding the chain - we need messages that were active AT this snaps
    const messagesResult = await this.pool.query(`
      WITH snapshot_chain AS (
        SELECT id, version FROM tm_snapshots
        WHERE chat_id = $1 AND version <= (SELECT version FROM tm_snapshots WHERE id = $2)
      )
      SELECT DISTINCT ON (mv.message_id) mv.*
      FROM tm_message_versions mv
      WHERE mv.snapshot_id IN (SELECT id FROM snapshot_chain)
        AND NOT mv.is_soft_deleted
      ORDER BY mv.message_id, mv.version DESC
    `, [chatId, snapshotId]);

    // Get files at this snapshot
    const filesResult = await this.pool.query(`
      WITH snapshot_chain AS (
        SELECT id, version FROM tm_snapshots
        WHERE chat_id = $1 AND version <= (SELECT version FROM tm_snapshots WHERE id = $2)
      )
      SELECT DISTINCT ON (mf.original_name) mf.*
      FROM tm_media_vault mf
      WHERE mf.snapshot_id IN (SELECT id FROM snapshot_chain)
        AND mf.status != 'soft_deleted'
      ORDER BY mf.original_name, mf.version DESC
    `, [chatId, snapshotId]);

    return {
      snapshot: this.mapSnapshotRow(snapshotResult.rows[0]),
      messages: messagesResult.rows.map(row => this.mapMessageVersionRow(row)),
      files: filesResult.rows.map(row => this.mapMediaVaultRow(row)),
    };
  }

  async getSnapshotsByDate(chatId: string, date: string, tenantId: string): Promise<TimeMachin
    await this.pool.query(`SET app.current_tenant_id = '${tenantId}'`);

    const result = await this.pool.query(`
      SELECT * FROM tm_snapshots
      WHERE chat_id = $1 AND DATE(snapshot_timestamp) = $2
      ORDER BY snapshot_timestamp ASC
    `, [chatId, date]);

    return result.rows.map(row => this.mapSnapshotRow(row));
  }

  //
  // RESTORE (One-click recovery)
  //

  async restore(request: RestoreRequest, userId: string, tenantId: string): Promise<RestoreResu
    const client = await this.pool.connect();

    try {
      await client.query('BEGIN');
      await client.query(`SET app.current_tenant_id = '${tenantId}'`);

      // Get target snapshot state
      const targetState = await this.getChatAtSnapshot(request.chatId, request.targetSnapshotI

      // Get current snapshot for logging
      const currentSnapshot = await this.getLatestSnapshot(client, request.chatId);

      let messagesRestored = 0;
      let filesRestored = 0;

      switch (request.scope) {
        case 'full_chat':
          // Restore all messages and files
          messagesRestored = await this.restoreMessages(client, targetState.messages, request.
          filesRestored = await this.restoreFiles(client, targetState.files, request.chatId, t
          break;

        case 'single_message':
          if (request.messageIds?.length) {
            const targetMessages = targetState.messages.filter(m => request.messageIds!.include
            messagesRestored = await this.restoreMessages(client, targetMessages, request.chatI
          }
          break;

        case 'single_file':
          if (request.fileIds?.length) {
            const targetFiles = targetState.files.filter(f => request.fileIds!.includes(f.id))
            filesRestored = await this.restoreFiles(client, targetFiles, request.chatId, tenan
          }
          break;

        case 'files_only':
```

```
        filesRestored = await this.restoreFiles(client, targetState.files, request.chatId, te
        break;
    }

    await client.query('COMMIT');

    // Create restore snapshot
    const newSnapshot = await this.createSnapshot({
      chatId: request.chatId,
      tenantId,
      trigger: 'restore_performed',
      triggerDescription: `Restored to version ${targetState.snapshot.version}`,
    });

    // Log the restore
    await this.pool.query(`
      INSERT INTO tm_restore_log (
        tenant_id, chat_id, user_id, from_snapshot_id, to_snapshot_id,
        scope, message_ids, file_ids, messages_restored, files_restored, reason
      ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11)
    `, [
      tenantId,
      request.chatId,
      userId,
      request.targetSnapshotId,
      newSnapshot.id,
      request.scope,
      request.messageIds || [],
      request.fileIds || [],
      messagesRestored,
      filesRestored,
      request.reason,
    ]);

    return {
      success: true,
      newSnapshotId: newSnapshot.id,
      messagesRestored,
      filesRestored,
      newVersion: newSnapshot.version,
      previousSnapshotId: currentSnapshot?.id || '',
    };
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
```

```
}

private async restoreMessages(
  client: PoolClient,
  messages: MessageVersion[],
  chatId: string,
  tenantId: string
): Promise<number> {
  // Deactivate current versions
  await client.query(`
    UPDATE tm_message_versions
    SET is_active = FALSE, superseded_at = NOW()
    WHERE message_id IN (
      SELECT DISTINCT message_id FROM tm_message_versions mv
      JOIN tm_snapshots s ON mv.snapshot_id = s.id
      WHERE s.chat_id = $1 AND mv.is_active = TRUE
    )
  `, [chatId]);

  // Get latest snapshot
  const snapshot = await this.getLatestSnapshot(client, chatId);

  // Insert restored versions as new active versions
  for (const msg of messages) {
    await client.query(`
      INSERT INTO tm_message_versions (
        tenant_id, message_id, snapshot_id, content, role, model_id,
        metadata, version, is_active, edit_reason, original_created_at
      ) VALUES ($1, $2, $3, $4, $5, $6, $7,
        (SELECT COALESCE(MAX(version), 0) + 1 FROM tm_message_versions WHERE message_id = $2
        TRUE, 'Restored from Time Machine', $8)
    `, [
      tenantId,
      msg.messageId,
      snapshot?.id,
      msg.content,
      msg.role,
      msg.modelId,
      JSON.stringify(msg.metadata || {}),
      msg.createdAt,
    ]);
  }

  return messages.length;
}

private async restoreFiles(
  client: PoolClient,
  files: MediaVaultFile[],
  chatId: string,
  tenantId: string
): Promise<number> {
  // Mark current files as soft deleted
  await client.query(`
    UPDATE tm_media_vault
    SET status = 'soft_deleted'
    WHERE chat_id = $1 AND status = 'active'
  `, [chatId]);

  // Get latest snapshot
  const snapshot = await this.getLatestSnapshot(client, chatId);

  // "Restore" files by creating new active versions pointing to same S3 objects
  for (const file of files) {
    await client.query(`
      INSERT INTO tm_media_vault (
        tenant_id, chat_id, message_id, snapshot_id, original_name, display_name,
        s3_bucket, s3_key, s3_version_id, mime_type, size_bytes, checksum_sha256,
        version, previous_version_id, source, status, extracted_text, ai_description
      ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12,
        (SELECT COALESCE(MAX(version), 0) + 1 FROM tm_media_vault WHERE chat_id = $2 AND orig
        $13, $14, 'active', $15, $16)
    `, [
      tenantId,
      chatId,
      file.messageId,
      snapshot?.id,
      file.originalName,
      file.displayName,
      file.s3Bucket,
      file.s3Key,
      file.s3VersionId,
      file.mimeType,
      file.sizeBytes,
      file.checksumSha256,
      file.id,
      file.source,
      file.extractedText,
      file.aiDescription,
    ]);
  }

  return files.length;
}

//
```

```
// SEARCH (Find that thing from 3 months ago)
//

async searchMessages(chatId: string, query: string, tenantId: string): Promise<MessageVersion
  await this.pool.query(`SET app.current_tenant_id = '${tenantId}'`);

  const result = await this.pool.query(`
    SELECT DISTINCT ON (mv.message_id) mv.*,
      ts_rank(to_tsvector('english', mv.content), plainto_tsquery('english', $2)) as rank
    FROM tm_message_versions mv
    JOIN tm_snapshots s ON mv.snapshot_id = s.id
    WHERE s.chat_id = $1
      AND to_tsvector('english', mv.content) @@ plainto_tsquery('english', $2)
    ORDER BY mv.message_id, rank DESC, mv.version DESC
    LIMIT 50
  `, [chatId, query]);

  return result.rows.map(row => this.mapMessageVersionRow(row));
}

async searchFiles(chatId: string, query: string, tenantId: string): Promise<MediaVaultFile[]
  await this.pool.query(`SET app.current_tenant_id = '${tenantId}'`);

  const result = await this.pool.query(`
    SELECT DISTINCT ON (original_name) *,
      ts_rank(
        to_tsvector('english', COALESCE(extracted_text, '') || ' ' || COALESCE(ai_descriptio
        plainto_tsquery('english', $2)
      ) as rank
    FROM tm_media_vault
    WHERE chat_id = $1
      AND (
        original_name ILIKE '%' || $2 || '%'
        OR to_tsvector('english', COALESCE(extracted_text, '') || ' ' || COALESCE(ai_descrip
           @@ plainto_tsquery('english', $2)
      )
    ORDER BY original_name, rank DESC, version DESC
    LIMIT 50
  `, [chatId, query]);

  return result.rows.map(row => this.mapMediaVaultRow(row));
}

//
// EXPORT
//

async createExportBundle(params: {
  chatId: string;
  tenantId: string;
  userId: string;
  format: ExportFormat;
  includeMedia: boolean;
  includeVersionHistory: boolean;
  fromSnapshotId?: string;
}): Promise<string> {
  // Get current snapshot
  const currentResult = await this.pool.query(`
    SELECT id FROM tm_snapshots
    WHERE chat_id = $1
    ORDER BY version DESC LIMIT 1
  `, [params.chatId]);

  const bundleId = uuid();

  await this.pool.query(`
    INSERT INTO tm_export_bundles (
      id, tenant_id, chat_id, user_id, from_snapshot_id, to_snapshot_id,
      format, include_media, include_version_history, status
    ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, 'pending')
  `, [
    bundleId,
    params.tenantId,
    params.chatId,
    params.userId,
    params.fromSnapshotId,
    currentResult.rows[0]?.id,
    params.format,
    params.includeMedia,
    params.includeVersionHistory,
  ]);

  // Trigger async export via SQS - see Section 33.5 for export queue handler
  // await sqs.send(new SendMessageCommand({
  //   QueueUrl: process.env.EXPORT_QUEUE_URL,
  //   MessageBody: JSON.stringify({ chatId, format, includeMedia }),
  // }));

  return bundleId;
}

//
// HELPERS
//

private async getLatestSnapshot(client: PoolClient, chatId: string): Promise<TimeMachineSnaps
```

```
    const result = await client.query(`
      SELECT * FROM tm_snapshots
      WHERE chat_id = $1
      ORDER BY version DESC LIMIT 1
    `, [chatId]);

    return result.rows[0] ? this.mapSnapshotRow(result.rows[0]) : null;
  }

  private mapSnapshotRow(row: any): TimeMachineSnapshot {
    return {
      id: row.id,
      chatId: row.chat_id,
      tenantId: row.tenant_id,
      version: row.version,
      timestamp: row.snapshot_timestamp,
      messageCount: row.message_count,
      fileCount: row.file_count,
      totalTokens: row.total_tokens,
      trigger: row.trigger,
      triggerDetails: {
        messageId: row.trigger_message_id,
        fileId: row.trigger_file_id,
        description: row.trigger_description,
      },
      previousSnapshotId: row.previous_snapshot_id,
      restoredFromSnapshotId: row.restored_from_snapshot_id,
      checksum: row.checksum,
      createdAt: row.created_at,
    };
  }

  private mapMessageVersionRow(row: any): MessageVersion {
    return {
      id: row.id,
      messageId: row.message_id,
      tenantId: row.tenant_id,
      snapshotId: row.snapshot_id,
      content: row.content,
      role: row.role,
      modelId: row.model_id,
      version: row.version,
      isActive: row.is_active,
      isSoftDeleted: row.is_soft_deleted,
      editReason: row.edit_reason,
      editedBy: row.edited_by,
      createdAt: row.created_at,
      supersededAt: row.superseded_at,
    };
  }

  private mapMediaVaultRow(row: any): MediaVaultFile {
    return {
      id: row.id,
      chatId: row.chat_id,
      tenantId: row.tenant_id,
      messageId: row.message_id,
      snapshotId: row.snapshot_id,
      originalName: row.original_name,
      displayName: row.display_name,
      s3Bucket: row.s3_bucket,
      s3Key: row.s3_key,
      s3VersionId: row.s3_version_id,
      mimeType: row.mime_type,
      sizeBytes: row.size_bytes,
      checksumSha256: row.checksum_sha256,
      thumbnailS3Key: row.thumbnail_s3_key,
      previewGenerated: row.preview_generated,
      version: row.version,
      previousVersionId: row.previous_version_id,
      source: row.source,
      status: row.status,
      extractedText: row.extracted_text,
      aiDescription: row.ai_description,
      createdAt: row.created_at,
      archivedAt: row.archived_at,
    };
  }
}
```

32.5 Complex API Handlers (Service Layer Exposure)

```
// packages/functions/src/handlers/thinktank/time-machine.handlers.ts

import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
import { TimeMachineService } from '../../services/time-machine.service';
import { pool } from '../../utils/db';
import { RestoreScope, ExportFormat } from '@radiant/shared';

const service = new TimeMachineService(pool);

const corsHeaders = {
  'Access-Control-Allow-Origin': '*',
  'Access-Control-Allow-Headers': 'Content-Type,Authorization',
  'Content-Type': 'application/json',
};

function getUserContext(event: APIGatewayProxyEvent) {
  return {
    userId: event.requestContext.authorizer?.claims?.sub,
    tenantId: event.requestContext.authorizer?.claims?.['custom:tenant_id'],
  };
}

//
// TIMELINE ENDPOINTS
//

// GET /api/thinktank/chats/:chatId/time-machine
export async function getTimeline(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult>
  try {
    const { tenantId } = getUserContext(event);
    const chatId = event.pathParameters?.chatId;

    if (!chatId) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'chatId rec
    }

    const timeline = await service.getTimeline(chatId, tenantId);

    return {
      statusCode: 200,
      headers: corsHeaders,
      body: JSON.stringify(timeline),
    };
  } catch (error: any) {
    console.error('getTimeline error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.message
  }
}

// GET /api/thinktank/chats/:chatId/time-machine/snapshots/:snapshotId
export async function getChatAtSnapshot(event: APIGatewayProxyEvent): Promise<APIGatewayProxyRe
  try {
    const { tenantId } = getUserContext(event);
    const chatId = event.pathParameters?.chatId;
    const snapshotId = event.pathParameters?.snapshotId;

    if (!chatId || !snapshotId) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'chatId and
    }
    const state = await service.getChatAtSnapshot(chatId, snapshotId, tenantId);

    return {
      statusCode: 200,
      headers: corsHeaders,
      body: JSON.stringify(state),
    };
  } catch (error: any) {
    console.error('getChatAtSnapshot error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.messag
  }
}

// GET /api/thinktank/chats/:chatId/time-machine/calendar/:date
export async function getSnapshotsByDate(event: APIGatewayProxyEvent): Promise<APIGatewayProxy
  try {
    const { tenantId } = getUserContext(event);
    const chatId = event.pathParameters?.chatId;
    const date = event.pathParameters?.date;  // YYYY-MM-DD

    if (!chatId || !date) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'chatId an
    }

    const snapshots = await service.getSnapshotsByDate(chatId, date, tenantId);

    return {
      statusCode: 200,
      headers: corsHeaders,
      body: JSON.stringify({ snapshots }),
    };
  } catch (error: any) {
    console.error('getSnapshotsByDate error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.messag
  }
}

//
// RESTORE ENDPOINTS
//

// POST /api/thinktank/chats/:chatId/time-machine/restore
export async function restoreFromSnapshot(event: APIGatewayProxyEvent): Promise<APIGatewayProx
  try {
    const { userId, tenantId } = getUserContext(event);
    const chatId = event.pathParameters?.chatId;
    const body = JSON.parse(event.body || '{}');
    if (!chatId || !body.snapshotId) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'chatId and
    }

    const result = await service.restore({
      chatId,
      targetSnapshotId: body.snapshotId,
      scope: (body.scope || 'full_chat') as RestoreScope,
      messageIds: body.messageIds,
      fileIds: body.fileIds,
      reason: body.reason,
    }, userId, tenantId);

    return {
      statusCode: 200,
      headers: corsHeaders,
      body: JSON.stringify(result),
    };
  } catch (error: any) {
    console.error('restoreFromSnapshot error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.message
  }
}

//
// MEDIA VAULT ENDPOINTS
//

// GET /api/thinktank/chats/:chatId/time-machine/files
export async function getFiles(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> {
  try {
    const { tenantId } = getUserContext(event);
    const chatId = event.pathParameters?.chatId;

    if (!chatId) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'chatId re
    }

    const result = await pool.query(`
      SELECT * FROM tm_files_with_versions
      WHERE chat_id = $1
      ORDER BY created_at DESC
    `, [chatId]);

    return {
      statusCode: 200,
      headers: corsHeaders,
      body: JSON.stringify({ files: result.rows }),
    };
  } catch (error: any) {
    console.error('getFiles error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.messag
  }
}

// GET /api/thinktank/chats/:chatId/time-machine/files/:fileName/versions
export async function getFileVersions(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResu
  try {
    const { tenantId } = getUserContext(event);
    const chatId = event.pathParameters?.chatId;
    const fileName = decodeURIComponent(event.pathParameters?.fileName || '');

    if (!chatId || !fileName) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'chatId anc
    }

    const versions = await service.getFileVersions(chatId, fileName);

    return {
      statusCode: 200,
      headers: corsHeaders,
      body: JSON.stringify({ versions }),
    };
  } catch (error: any) {
    console.error('getFileVersions error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.messag
  }
}

// GET /api/thinktank/files/:fileId/download
export async function downloadFile(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult
  try {
    const fileId = event.pathParameters?.fileId;

    if (!fileId) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'fileId rec
    }

    const url = await service.getFileDownloadUrl(fileId);

    return {
      statusCode: 302,
      headers: { ...corsHeaders, Location: url },
      body: '',
    };
  } catch (error: any) {
    console.error('downloadFile error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.messag
  }
}

//
// SEARCH ENDPOINTS
//

// GET /api/thinktank/chats/:chatId/time-machine/search
export async function search(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> {
  try {
    const { tenantId } = getUserContext(event);
    const chatId = event.pathParameters?.chatId;
    const query = event.queryStringParameters?.q;
    const type = event.queryStringParameters?.type || 'all';  // 'messages', 'files', 'all'

    if (!chatId || !query) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'chatId an
    }

    const results: { messages?: any[]; files?: any[] } = {};

    if (type === 'all' || type === 'messages') {
      results.messages = await service.searchMessages(chatId, query, tenantId);
    }

    if (type === 'all' || type === 'files') {
      results.files = await service.searchFiles(chatId, query, tenantId);
    }

    return {
      statusCode: 200,
      headers: corsHeaders,
      body: JSON.stringify(results),
    };
  } catch (error: any) {
    console.error('search error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.messag
  }
}

//
// EXPORT ENDPOINTS
//

// POST /api/thinktank/chats/:chatId/time-machine/export
export async function createExport(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult
```

```
  try {
    const { userId, tenantId } = getUserContext(event);
    const chatId = event.pathParameters?.chatId;
    const body = JSON.parse(event.body || '{}');

    if (!chatId) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'chatId re
    }

    const bundleId = await service.createExportBundle({
      chatId,
      tenantId,
      userId,
      format: (body.format || 'zip') as ExportFormat,
      includeMedia: body.includeMedia !== false,
      includeVersionHistory: body.includeVersionHistory === true,
      fromSnapshotId: body.fromSnapshotId,
    });

    return {
      statusCode: 202,
      headers: corsHeaders,
      body: JSON.stringify({
        bundleId,
        status: 'pending',
        message: 'Export is being prepared. Check status at /api/thinktank/exports/:bundleId',
      }),
    };
  } catch (error: any) {
    console.error('createExport error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.messag
  }
}

// GET /api/thinktank/exports/:bundleId
export async function getExportStatus(event: APIGatewayProxyEvent): Promise<APIGatewayProxyRes
  try {
    const bundleId = event.pathParameters?.bundleId;

    if (!bundleId) {
      return { statusCode: 400, headers: corsHeaders, body: JSON.stringify({ error: 'bundleId 
    }

    const result = await pool.query(`
      SELECT * FROM tm_export_bundles WHERE id = $1
    `, [bundleId]);

    if (!result.rows[0]) {
      return { statusCode: 404, headers: corsHeaders, body: JSON.stringify({ error: 'Export no
    }

    const bundle = result.rows[0];

    // If ready, generate download URL
    let downloadUrl: string | undefined;
    if (bundle.status === 'ready' && bundle.s3_key) {
      downloadUrl = await service.getFileDownloadUrl(bundle.s3_key);
    }

    return {
      statusCode: 200,
      headers: corsHeaders,
      body: JSON.stringify({
        bundleId: bundle.id,
        status: bundle.status,
        format: bundle.format,
        sizeBytes: bundle.size_bytes,
        downloadUrl,
        expiresAt: bundle.expires_at,
        createdAt: bundle.created_at,
        completedAt: bundle.completed_at,
        errorMessage: bundle.error_message,
      }),
    };
  } catch (error: any) {
    console.error('getExportStatus error:', error);
    return { statusCode: 500, headers: corsHeaders, body: JSON.stringify({ error: error.messag
  }
}
```

32.6 API Routes Configuration

```
// packages/functions/src/routes/time-machine.routes.ts

import { Router } from './router';
import * as handlers from '../handlers/thinktank/time-machine.handlers';

export function registerTimeMachineRoutes(router: Router) {
  // Timeline
  router.get('/api/thinktank/chats/:chatId/time-machine', handlers.getTimeline);
  router.get('/api/thinktank/chats/:chatId/time-machine/snapshots/:snapshotId', handlers.getCha
  router.get('/api/thinktank/chats/:chatId/time-machine/calendar/:date', handlers.getSnapshots

  // Restore
  router.post('/api/thinktank/chats/:chatId/time-machine/restore', handlers.restoreFromSnapsho

  // Files
  router.get('/api/thinktank/chats/:chatId/time-machine/files', handlers.getFiles);
  router.get('/api/thinktank/chats/:chatId/time-machine/files/:fileName/versions', handlers.ge
  router.get('/api/thinktank/files/:fileId/download', handlers.downloadFile);

  // Search
  router.get('/api/thinktank/chats/:chatId/time-machine/search', handlers.search);

  // Export
  router.post('/api/thinktank/chats/:chatId/time-machine/export', handlers.createExport);
  router.get('/api/thinktank/exports/:bundleId', handlers.getExportStatus);
}
```