🚀 Phase 3 Implementation Guide

Advanced Features & Enterprise-Ready Platform

📋 Phase 3 Overview

Timeline: Week 7-10
Total Effort: 160 hours
Expected Impact: Enterprise-ready platform with scalability

Advanced Improvements:

1. Multi-Tenancy Support
2. Usage Analytics & Billing System
3. Automated Data Archival
4. Vector Database Integration (Qdrant)
5. Distributed Tracing & Advanced Monitoring

1️⃣ Multi-Tenancy Support

🎯 Goal

Enable multiple organizations/teams to use platform with complete data isolation

📊 Database Schema

File: `migrations/phase3_multi_tenancy.sql`

```
-- Create Tenant table
CREATE TABLE tenants (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL UNIQUE,
    domain VARCHAR(255) NOT NULL UNIQUE,
    logo_url TEXT,
    stripe_customer_id VARCHAR(255),
    subscription_tier VARCHAR(50),  -- free, pro, enterprise
    storage_quota BIGINT DEFAULT 1099511627776,  -- 1TB
    api_quota INTEGER DEFAULT 100000,  -- 100k requests/month
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW(),
    INDEX idx_tenants_domain (domain),
    INDEX idx_tenants_is_active (is_active)
);
```

```
-- Modify Users table to add tenant_id
ALTER TABLE users ADD COLUMN tenant_id UUID NOT NULL REFERENCES tenants(id) ON DELETE CAS
ALTER TABLE users ADD COLUMN role VARCHAR(50) DEFAULT 'member';  -- admin, editor, viewe
CREATE INDEX idx_users_tenant_id ON users(tenant_id);
CREATE INDEX idx_users_tenant_email ON users(tenant_id, email);

-- Add tenant_id to Files table
ALTER TABLE files ADD COLUMN tenant_id UUID NOT NULL;
ALTER TABLE files ADD CONSTRAINT fk_files_tenant FOREIGN KEY (tenant_id) REFERENCES tenan
CREATE INDEX idx_files_tenant_status ON files(tenant_id, status);
CREATE INDEX idx_files_tenant_created ON files(tenant_id, created_at DESC);

-- Add tenant_id to Projects table
ALTER TABLE projects ADD COLUMN tenant_id UUID NOT NULL;
ALTER TABLE projects ADD CONSTRAINT fk_projects_tenant FOREIGN KEY (tenant_id) REFERENCES
CREATE INDEX idx_projects_tenant ON projects(tenant_id);

-- Add tenant_id to Analysis tables
ALTER TABLE file_analysis ADD COLUMN tenant_id UUID NOT NULL;
ALTER TABLE file_analysis ADD CONSTRAINT fk_analysis_tenant FOREIGN KEY (tenant_id) REFER
CREATE INDEX idx_analysis_tenant ON file_analysis(tenant_id);

-- Create TenantSettings table for per-tenant configuration
CREATE TABLE tenant_settings (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID NOT NULL UNIQUE REFERENCES tenants(id) ON DELETE CASCADE,
    max_file_size BIGINT DEFAULT 5368709120,  -- 5GB
    allowed_file_types TEXT[],
    webhook_url TEXT,
    webhook_events TEXT[],
    custom_branding JSONB,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Create TenantInvitations table
CREATE TABLE tenant_invitations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID NOT NULL REFERENCES tenants(id) ON DELETE CASCADE,
    email VARCHAR(255) NOT NULL,
    role VARCHAR(50) DEFAULT 'member',
    token VARCHAR(255) NOT NULL UNIQUE,
    expires_at TIMESTAMP NOT NULL,
    accepted_at TIMESTAMP,
    created_at TIMESTAMP DEFAULT NOW()
);

-- Create AuditLog table (already in Phase 2 but with tenant_id)
ALTER TABLE audit_logs ADD COLUMN tenant_id UUID;
ALTER TABLE audit_logs ADD CONSTRAINT fk_audit_tenant FOREIGN KEY (tenant_id) REFERENCES
CREATE INDEX idx_audit_tenant_timestamp ON audit_logs(tenant_id, timestamp DESC);
```

🏢 Multi-Tenancy Models

File: `python-services/api/models/tenant.py`

```
from sqlalchemy import Column, String, Integer, Boolean, DateTime, JSON, ForeignKey
from sqlalchemy.dialects.postgresql import UUID
from sqlalchemy.orm import relationship
from datetime import datetime
import uuid

class Tenant(Base):
    __tablename__ = "tenants"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    name = Column(String(255), unique=True, index=True, nullable=False)
    domain = Column(String(255), unique=True, index=True, nullable=False)
    logo_url = Column(String(500))
    stripe_customer_id = Column(String(255))
    subscription_tier = Column(String(50), default="free")
    storage_quota = Column(Integer, default=1099511627776)  # 1TB
    api_quota = Column(Integer, default=100000)
    is_active = Column(Boolean, default=True, index=True)
    created_at = Column(DateTime, default=datetime.now)
    updated_at = Column(DateTime, default=datetime.now, onupdate=datetime.now)

    # Relationships
    users = relationship("User", back_populates="tenant", cascade="all, delete-orphan")
    projects = relationship("Project", back_populates="tenant", cascade="all, delete-orph
    files = relationship("File", back_populates="tenant", cascade="all, delete-orphan")
    settings = relationship("TenantSettings", back_populates="tenant", uselist=False)

class TenantSettings(Base):
    __tablename__ = "tenant_settings"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    tenant_id = Column(UUID(as_uuid=True), ForeignKey("tenants.id", ondelete="CASCADE"),
    max_file_size = Column(Integer, default=5368709120)  # 5GB
    allowed_file_types = Column(JSON, default=["dwg", "rvt", "ifc", "pdf"])
    webhook_url = Column(String(500))
    webhook_events = Column(JSON, default=["file.uploaded", "analysis.completed"])
    custom_branding = Column(JSON)
    created_at = Column(DateTime, default=datetime.now)
    updated_at = Column(DateTime, default=datetime.now, onupdate=datetime.now)

    tenant = relationship("Tenant", back_populates="settings")

class User(Base):
    __tablename__ = "users"

    # ... existing columns ...
    tenant_id = Column(UUID(as_uuid=True), ForeignKey("tenants.id", ondelete="CASCADE"),
    role = Column(String(50), default="member")  # admin, editor, viewer

    tenant = relationship("Tenant", back_populates="users")

# Updated File model
```

```
class File(Base):
    __tablename__ = "files"

    # ... existing columns ...
    tenant_id = Column(UUID(as_uuid=True), ForeignKey("tenants.id", ondelete="CASCADE"),

    tenant = relationship("Tenant", back_populates="files")
```

🔧 Multi-Tenancy Middleware

File: `python-services/api/middleware/tenant_middleware.py`

```
from fastapi import Request, HTTPException, Depends
from sqlalchemy.orm import Session
from typing import Optional
import logging

logger = logging.getLogger(__name__)

class TenantMiddleware:
    @staticmethod
    def extract_tenant_id(request: Request) -&gt; Optional[str]:
        """Extract tenant ID from request (subdomain, header, or domain)"""

        # Method 1: X-Tenant-ID header
        tenant_id = request.headers.get("X-Tenant-ID")
        if tenant_id:
            return tenant_id

        # Method 2: Subdomain extraction
        host = request.headers.get("Host", "").lower()
        if host:
            parts = host.split(".")
            if len(parts) &gt;= 3:  # subdomain.example.com
                subdomain = parts[0]
                if subdomain not in ["www", "api", "admin"]:
                    return subdomain

        # Method 3: From JWT token in Authorization header
        auth_header = request.headers.get("Authorization", "")
        if auth_header.startswith("Bearer "):
            token = auth_header.replace("Bearer ", "")
            # Decode JWT and extract tenant_id
            try:
                payload = jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
                return payload.get("tenant_id")
            except:
                pass

        return None

async def get_current_tenant(
    request: Request,
    db: Session = Depends(get_db)
) -&gt; Tenant:
```

```
    """Get current tenant from request"""

    tenant_id = TenantMiddleware.extract_tenant_id(request)

    if not tenant_id:
        raise HTTPException(status_code=400, detail="Tenant not specified")

    tenant = db.query(Tenant).filter(
        Tenant.id == tenant_id,
        Tenant.is_active == True
    ).first()

    if not tenant:
        raise HTTPException(status_code=404, detail="Tenant not found")

    request.scope["tenant_id"] = tenant.id
    request.scope["tenant"] = tenant

    return tenant

async def tenant_isolation_middleware(request: Request, call_next):
    """Middleware to enforce tenant isolation"""

    tenant_id = TenantMiddleware.extract_tenant_id(request)

    if tenant_id:
        request.scope["tenant_id"] = tenant_id

    response = await call_next(request)
    return response

# Add to FastAPI app
app.add_middleware(tenant_isolation_middleware)
```

Tenant-Aware Routes

File: `python-services/api/routes/files_multitenant.py`

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from typing import List

router = APIRouter(prefix="/api", tags=["files"])

@router.get("/files")
async def list_tenant_files(
    tenant: Tenant = Depends(get_current_tenant),
    db: Session = Depends(get_db),
    skip: int = 0,
    limit: int = 50
) -&gt; List[dict]:
    """List files for current tenant only"""

    # IMPORTANT: Always filter by tenant_id
    files = db.query(File).filter(
```

```
        File.tenant_id == tenant.id
    ).offset(skip).limit(limit).all()

    return files

@router.post("/files/upload")
async def upload_file(
    file: UploadFile,
    tenant: Tenant = Depends(get_current_tenant),
    db: Session = Depends(get_db)
):
    """Upload file to current tenant"""

    # Check tenant storage quota
    tenant_usage = db.query(func.sum(File.file_size)).filter(
        File.tenant_id == tenant.id
    ).scalar() or 0

    if tenant_usage + file.size &gt; tenant.storage_quota:
        raise HTTPException(
            status_code=429,
            detail="Storage quota exceeded"
        )

    # Create file record
    file_record = File(
        tenant_id=tenant.id,  # ← MUST set tenant_id
        user_id=user_id,
        filename=file.filename,
        file_size=file.size
    )

    db.add(file_record)
    db.commit()

    return {"file_id": file_record.id}

@router.get("/analytics/dashboard")
async def get_tenant_analytics(
    tenant: Tenant = Depends(get_current_tenant),
    db: Session = Depends(get_db)
):
    """Get analytics for current tenant only"""

    files = db.query(File).filter(
        File.tenant_id == tenant.id
    ).all()

    return {
        "total_files": len(files),
        "storage_used": sum(f.file_size for f in files),
        "storage_quota": tenant.storage_quota,
        "api_usage": get_tenant_api_usage(db, tenant.id),
        "subscription_tier": tenant.subscription_tier
    }
```

⚛ Frontend Tenant Handling

File: `web-react/src/hooks/useTenant.js`

```
import { useContext, createContext } from 'react';
import axios from 'axios';

const TenantContext = createContext();

export const TenantProvider = ({ children }) =&gt; {
  const [tenant, setTenant] = React.useState(null);
  const [loading, setLoading] = React.useState(true);

  React.useEffect(() =&gt; {
    fetchCurrentTenant();
  }, []);

  const fetchCurrentTenant = async () =&gt; {
    try {
      // Extract tenant from subdomain or header
      const subdomain = window.location.hostname.split('.')[0];

      const response = await axios.get('/api/tenant', {
        headers: {
          'X-Tenant-ID': subdomain
        }
      });

      setTenant(response.data);
    } catch (error) {
      console.error('Failed to load tenant:', error);
    } finally {
      setLoading(false);
    }
  };

  return (
    &lt;TenantContext.Provider value={{ tenant, loading }}&gt;
      {children}
    &lt;/TenantContext.Provider&gt;
  );
};

export const useTenant = () =&gt; {
  const context = useContext(TenantContext);
  if (!context) {
    throw new Error('useTenant must be used within TenantProvider');
  }
  return context;
};

// Usage in components
export const FileUpload = () =&gt; {
  const { tenant } = useTenant();

  const handleUpload = async (file) =&gt; {
```

```
    await axios.post('/api/files/upload', { file }, {
      headers: {
        'X-Tenant-ID': tenant.id
      }
    });
  };

  return (
    <div>
      <p>Uploading to: {tenant.name}</p>
      {/* Upload UI */}
    </div>
  );
};
```

2 Usage Analytics & Billing System

Goal

Track usage metrics and implement usage-based billing

Usage Tracking Models

File: `python-services/api/models/billing.py`

```
from sqlalchemy import Column, String, Integer, Float, DateTime, JSON, ForeignKey
from sqlalchemy.dialects.postgresql import UUID
from enum import Enum
from datetime import datetime
import uuid

class UsageMetric(Base):
    __tablename__ = "usage_metrics"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    tenant_id = Column(UUID(as_uuid=True), ForeignKey("tenants.id", ondelete="CASCADE"),
    metric_date = Column(DateTime, index=True)

    # Usage metrics
    files_uploaded = Column(Integer, default=0)
    files_processed = Column(Integer, default=0)
    api_calls = Column(Integer, default=0)
    storage_used = Column(Integer, default=0)  # bytes
    cost_estimated = Column(Float, default=0.0)
    carbon_calculated = Column(Float, default=0.0)

    # Timestamps
    created_at = Column(DateTime, default=datetime.now)
    updated_at = Column(DateTime, default=datetime.now, onupdate=datetime.now)

    __table_args__ = (
        Index('idx_usage_tenant_date', 'tenant_id', 'metric_date'),
```

```
    )

class Invoice(Base):
    __tablename__ = "invoices"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    tenant_id = Column(UUID(as_uuid=True), ForeignKey("tenants.id", ondelete="CASCADE"),
    stripe_invoice_id = Column(String(255), unique=True)

    # Billing period
    billing_period_start = Column(DateTime)
    billing_period_end = Column(DateTime)

    # Charges
    base_price = Column(Float)  # Subscription price
    usage_charges = Column(Float)  # Overage charges
    discount = Column(Float, default=0.0)
    total_amount = Column(Float)
    tax_amount = Column(Float, default=0.0)

    # Status
    status = Column(String(50))  # draft, sent, paid, overdue
    paid_at = Column(DateTime)
    due_date = Column(DateTime)

    # Metadata
    line_items = Column(JSON)
    notes = Column(String(500))

    created_at = Column(DateTime, default=datetime.now)
    updated_at = Column(DateTime, default=datetime.now, onupdate=datetime.now)

class BillingPlan(Base):
    __tablename__ = "billing_plans"

    id = Column(String(50), primary_key=True)  # free, pro, enterprise
    name = Column(String(100))
    monthly_price = Column(Float)

    # Limits
    file_processing_limit = Column(Integer)  # files/month
    storage_limit = Column(Integer)  # bytes
    api_calls_limit = Column(Integer)  # calls/month

    # Overage pricing
    cost_per_extra_file = Column(Float, default=0.50)
    cost_per_gb_storage = Column(Float, default=10.0)
    cost_per_1k_api_calls = Column(Float, default=0.10)

    is_active = Column(Boolean, default=True)
    created_at = Column(DateTime, default=datetime.now)

# Insert pricing plans
INSERT INTO billing_plans VALUES
('free', 'Free', 0.0, 100, 10737418240, 10000, 0.50, 10.0, 0.10, true, NOW()),
```

```
('pro', 'Professional', 99.0, 1000, 107374182400, 100000, 0.50, 10.0, 0.10, true, NOW()),
('enterprise', 'Enterprise', 999.0, 10000, 1099511627776, 1000000, 0.25, 5.0, 0.05, true,
```

📊 Usage Tracking Service

File: `python-services/api/services/usage_tracker.py`

```
from sqlalchemy.orm import Session
from sqlalchemy import func, text
from datetime import datetime, timedelta
import logging

logger = logging.getLogger(__name__)

class UsageTracker:
    @staticmethod
    async def track_file_upload(db: Session, tenant_id: str, file_size: int):
        """Track file upload usage"""

        today = datetime.now().date()

        metric = db.query(UsageMetric).filter(
            UsageMetric.tenant_id == tenant_id,
            func.date(UsageMetric.metric_date) == today
        ).first()

        if not metric:
            metric = UsageMetric(
                tenant_id=tenant_id,
                metric_date=datetime.now()
            )
            db.add(metric)

        metric.files_uploaded += 1
        metric.storage_used += file_size
        db.commit()

        # Check if over quota
        await UsageTracker.check_usage_limits(db, tenant_id)

    @staticmethod
    async def track_api_call(db: Session, tenant_id: str):
        """Track API call usage"""

        today = datetime.now().date()

        metric = db.query(UsageMetric).filter(
            UsageMetric.tenant_id == tenant_id,
            func.date(UsageMetric.metric_date) == today
        ).first()

        if metric:
            metric.api_calls += 1
            db.commit()
```

```
    @staticmethod
    async def track_file_processing(db: Session, tenant_id: str):
        """Track file processing"""

        today = datetime.now().date()

        metric = db.query(UsageMetric).filter(
            UsageMetric.tenant_id == tenant_id,
            func.date(UsageMetric.metric_date) == today
        ).first()

        if metric:
            metric.files_processed += 1
            db.commit()

    @staticmethod
    async def check_usage_limits(db: Session, tenant_id: str):
        """Check if tenant exceeded usage limits"""

        tenant = db.query(Tenant).get(tenant_id)
        plan = db.query(BillingPlan).filter(
            BillingPlan.id == tenant.subscription_tier
        ).first()

        # Get current month usage
        today = datetime.now().date()
        first_of_month = today.replace(day=1)

        month_usage = db.query(UsageMetric).filter(
            UsageMetric.tenant_id == tenant_id,
            UsageMetric.metric_date &gt;= first_of_month
        ).all()

        total_files = sum(m.files_processed for m in month_usage)
        total_storage = sum(m.storage_used for m in month_usage)
        total_api_calls = sum(m.api_calls for m in month_usage)

        warnings = []

        if total_files &gt; plan.file_processing_limit * 0.9:
            warnings.append("Approaching file processing limit")

        if total_storage &gt; plan.storage_limit * 0.9:
            warnings.append("Approaching storage limit")

        if total_api_calls &gt; plan.api_calls_limit * 0.9:
            warnings.append("Approaching API call limit")

        if warnings:
            logger.warning(f"Tenant {tenant_id} usage warnings: {warnings}")
            # Send email to tenant
            await send_usage_warning_email(tenant, warnings)

class BillingService:
    @staticmethod
    async def generate_monthly_invoices(db: Session):
```

```
        """Generate invoices for all tenants"""

        tenants = db.query(Tenant).filter(Tenant.is_active == True).all()

        for tenant in tenants:
            # Get previous month usage
            today = datetime.now().date()
            first_of_month = today.replace(day=1)
            last_month_end = first_of_month - timedelta(days=1)
            last_month_start = last_month_end.replace(day=1)

            usage = db.query(UsageMetric).filter(
                UsageMetric.tenant_id == tenant.id,
                UsageMetric.metric_date &gt;= last_month_start,
                UsageMetric.metric_date &lt;= last_month_end
            ).all()

            # Calculate charges
            plan = db.query(BillingPlan).filter(
                BillingPlan.id == tenant.subscription_tier
            ).first()

            base_price = plan.monthly_price

            # Calculate overage charges
            total_files = sum(u.files_processed for u in usage)
            total_storage = sum(u.storage_used for u in usage) / (1024**3)  # Convert to
            total_api_calls = sum(u.api_calls for u in usage)

            files_overage = max(0, total_files - plan.file_processing_limit)
            storage_overage = max(0, total_storage - plan.storage_limit / (1024**3))
            api_calls_overage = max(0, total_api_calls - plan.api_calls_limit)

            file_charges = files_overage * plan.cost_per_extra_file
            storage_charges = storage_overage * plan.cost_per_gb_storage
            api_charges = (api_calls_overage / 1000) * plan.cost_per_1k_api_calls

            usage_charges = file_charges + storage_charges + api_charges
            tax = (base_price + usage_charges) * 0.1  # 10% tax
            total = base_price + usage_charges + tax

            # Create invoice
            invoice = Invoice(
                tenant_id=tenant.id,
                billing_period_start=last_month_start,
                billing_period_end=last_month_end,
                base_price=base_price,
                usage_charges=usage_charges,
                tax_amount=tax,
                total_amount=total,
                status="draft",
                due_date=today + timedelta(days=30),
                line_items=[
                    {"description": f"Base subscription", "amount": base_price},
                    {"description": f"{files_overage} extra files", "amount": file_charge
                    {"description": f"{storage_overage:.2f}GB overage", "amount": storage
```

```
                    {"description": f"{api_calls_overage:,} extra API calls", "amount": a
                ]
            )

            db.add(invoice)

        db.commit()

        logger.info(f"Generated {len(tenants)} invoices")

    @staticmethod
    async def send_invoice_to_stripe(db: Session, invoice_id: str):
        """Send invoice to Stripe for payment processing"""

        import stripe

        invoice = db.query(Invoice).get(invoice_id)
        tenant = db.query(Tenant).get(invoice.tenant_id)

        stripe.api_key = STRIPE_SECRET_KEY

        try:
            stripe_invoice = stripe.Invoice.create(
                customer=tenant.stripe_customer_id,
                amount_due=int(invoice.total_amount * 100),  # Convert to cents
                currency="usd",
                metadata={
                    "invoice_id": str(invoice_id),
                    "tenant_id": str(invoice.tenant_id)
                }
            )

            invoice.stripe_invoice_id = stripe_invoice.id
            invoice.status = "sent"
            db.commit()

            logger.info(f"Sent invoice {invoice_id} to Stripe")

        except Exception as e:
            logger.error(f"Failed to send invoice to Stripe: {str(e)}")
            raise

# Routes
@router.get("/api/usage/current")
async def get_current_usage(
    tenant: Tenant = Depends(get_current_tenant),
    db: Session = Depends(get_db)
):
    """Get current month usage for tenant"""

    today = datetime.now().date()
    first_of_month = today.replace(day=1)

    usage = db.query(UsageMetric).filter(
        UsageMetric.tenant_id == tenant.id,
        UsageMetric.metric_date &gt;= first_of_month
```

```
    ).all()

    plan = db.query(BillingPlan).filter(
        BillingPlan.id == tenant.subscription_tier
    ).first()

    total_files = sum(u.files_processed for u in usage)
    total_storage = sum(u.storage_used for u in usage)
    total_api_calls = sum(u.api_calls for u in usage)

    return {
        "plan": tenant.subscription_tier,
        "current_period": {
            "start": first_of_month.isoformat(),
            "end": today.isoformat()
        },
        "usage": {
            "files": {
                "used": total_files,
                "limit": plan.file_processing_limit,
                "percentage": (total_files / plan.file_processing_limit * 100) if plan.fi
            },
            "storage": {
                "used": total_storage / (1024**3),  # GB
                "limit": plan.storage_limit / (1024**3),
                "percentage": (total_storage / plan.storage_limit * 100) if plan.storage_
            },
            "api_calls": {
                "used": total_api_calls,
                "limit": plan.api_calls_limit,
                "percentage": (total_api_calls / plan.api_calls_limit * 100) if plan.api_
            }
        }
    }

@router.get("/api/invoices")
async def list_invoices(
    tenant: Tenant = Depends(get_current_tenant),
    db: Session = Depends(get_db)
):
    """List invoices for tenant"""

    invoices = db.query(Invoice).filter(
        Invoice.tenant_id == tenant.id
    ).order_by(Invoice.created_at.desc()).all()

    return invoices

@router.post("/admin/generate-invoices")
async def trigger_invoice_generation(
    db: Session = Depends(get_db),
    admin: User = Depends(require_admin)
):
    """Admin endpoint to generate monthly invoices"""

    await BillingService.generate_monthly_invoices(db)
```

```
    return {"status": "invoices generated"}
```

3 Automated Data Archival

Goal

Automatically archive old files to cold storage (S3 Glacier)

Archival Service

File: `python-services/api/services/archival.py`

```
import boto3
import logging
from datetime import datetime, timedelta
from sqlalchemy.orm import Session
from sqlalchemy import func
import os

logger = logging.getLogger(__name__)

s3_client = boto3.client(
    's3',
    aws_access_key_id=os.getenv('AWS_ACCESS_KEY_ID'),
    aws_secret_access_key=os.getenv('AWS_SECRET_ACCESS_KEY')
)

class ArchivalService:
    ARCHIVE_CONFIG = {
        'completed_files': {'days': 365, 'storage_class': 'GLACIER'},
        'failed_files': {'days': 90, 'storage_class': 'STANDARD_IA'},
        'temp_files': {'days': 7, 'storage_class': 'GLACIER'},
        'logs': {'days': 30, 'storage_class': 'STANDARD_IA'}
    }

    @staticmethod
    async def archive_old_files(db: Session):
        """Archive files to S3 Glacier based on age and status"""

        for file_type, config in ArchivalService.ARCHIVE_CONFIG.items():
            cutoff_date = datetime.now() - timedelta(days=config['days'])

            if file_type == 'completed_files':
                files = db.query(File).filter(
                    File.created_at &lt; cutoff_date,
                    File.status == 'completed',
                    File.archived == False
                ).all()

            elif file_type == 'failed_files':
                files = db.query(File).filter(
```

```
                File.created_at &lt; cutoff_date,
                File.status == 'failed',
                File.archived == False
            ).all()

        elif file_type == 'temp_files':
            files = db.query(File).filter(
                File.created_at &lt; cutoff_date,
                File.is_temporary == True,
                File.archived == False
            ).all()

        archived_count = 0

        for file in files:
            try:
                await ArchivalService.archive_file(
                    db, file, config['storage_class']
                )
                archived_count += 1
            except Exception as e:
                logger.error(f"Failed to archive {file.id}: {str(e)}")

        logger.info(f"Archived {archived_count} {file_type}")

@staticmethod
async def archive_file(db: Session, file: File, storage_class: str):
    """Archive single file to S3"""

    try:
        # Read file from local storage
        with open(file.path, 'rb') as f:
            file_content = f.read()

        # Upload to S3
        archive_key = f"archive/{file.tenant_id}/{file.id}/{file.filename}"

        s3_client.put_object(
            Bucket=os.getenv('AWS_ARCHIVE_BUCKET', 'construction-ai-archive'),
            Key=archive_key,
            Body=file_content,
            StorageClass=storage_class,
            Metadata={
                'original_path': file.path,
                'tenant_id': str(file.tenant_id),
                'file_id': str(file.id),
                'archived_date': datetime.now().isoformat()
            }
        )

        # Update database
        file.archived = True
        file.archive_location = f"s3://{os.getenv('AWS_ARCHIVE_BUCKET')}/{archive_key
        file.archived_at = datetime.now()

        # Delete from local storage
```

```
            try:
                os.remove(file.path)
            except OSError:
                logger.warning(f"Could not delete {file.path}")

            db.commit()

            logger.info(f"Archived file {file.id} to S3")

        except Exception as e:
            logger.error(f"Archival failed for {file.id}: {str(e)}")
            raise

    @staticmethod
    async def restore_file(db: Session, file_id: str, restore_duration_days: int = 7):
        """Restore archived file from S3 Glacier"""

        file = db.query(File).get(file_id)

        if not file or not file.archived:
            raise HTTPException(status_code=404, detail="File not found or not archived")

        # Initiate restore from Glacier
        archive_key = file.archive_location.replace(
            f"s3://{os.getenv('AWS_ARCHIVE_BUCKET')}/", ""
        )

        try:
            s3_client.restore_object(
                Bucket=os.getenv('AWS_ARCHIVE_BUCKET'),
                Key=archive_key,
                RestoreRequest={
                    'Days': restore_duration_days,
                    'GlacierJobParameters': {'Tier': 'Standard'}
                }
            )

            file.restore_requested_at = datetime.now()
            db.commit()

            logger.info(f"Initiated restore for file {file_id}")

            return {
                "status": "restore_initiated",
                "file_id": file_id,
                "estimated_ready": (datetime.now() + timedelta(hours=4)).isoformat()
            }

        except Exception as e:
            logger.error(f"Restore failed for {file_id}: {str(e)}")
            raise

# Scheduled task (using APScheduler)
from apscheduler.schedulers.background import BackgroundScheduler

scheduler = BackgroundScheduler()
```

```
@scheduler.scheduled_job('cron', hour=2, minute=0)
async def nightly_archival():
    """Run archival job nightly at 2 AM"""
    db = SessionLocal()
    try:
        await ArchivalService.archive_old_files(db)
    finally:
        db.close()

scheduler.start()

# Routes
@router.post("/admin/archive-files")
async def trigger_archival(
    db: Session = Depends(get_db),
    admin: User = Depends(require_admin)
):
    """Admin endpoint to trigger file archival"""

    await ArchivalService.archive_old_files(db)

    return {"status": "archival completed"}

@router.post("/files/{file_id}/restore")
async def restore_archived_file(
    file_id: str,
    restore_days: int = 7,
    tenant: Tenant = Depends(get_current_tenant),
    db: Session = Depends(get_db)
):
    """Restore archived file from Glacier"""

    file = db.query(File).filter(
        File.id == file_id,
        File.tenant_id == tenant.id
    ).first()

    if not file:
        raise HTTPException(status_code=404, detail="File not found")

    result = await ArchivalService.restore_file(db, file_id, restore_days)

    return result
```

4️ Vector Database Integration (Qdrant)

Goal

Implement similarity search for cost estimation and material matching

Qdrant Integration

File: `python-services/api/services/vector_search.py`

```
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, PointStruct
from sentence_transformers import SentenceTransformer
import logging
from typing import List, Dict
import numpy as np

logger = logging.getLogger(__name__)

class VectorSearchService:
    def __init__(self):
        self.client = QdrantClient(url="http://qdrant:6333")
        self.embedding_model = SentenceTransformer('all-MiniLM-L6-v2')
        self.collection_name = "cost_estimates"
        self.embedding_dim = 384

        self._ensure_collection()

    def _ensure_collection(self):
        """Create collection if it doesn't exist"""

        try:
            self.client.get_collection(self.collection_name)
        except:
            self.client.create_collection(
                collection_name=self.collection_name,
                vectors_config=VectorParams(
                    size=self.embedding_dim,
                    distance=Distance.COSINE
                )
            )
            logger.info(f"Created Qdrant collection: {self.collection_name}")

    def _generate_embedding(self, text: str) -&gt; List[float]:
        """Generate embedding for text"""

        embedding = self.embedding_model.encode(text)
        return embedding.tolist()

    async def index_cost_estimate(
        self,
        estimate_id: str,
        materials: Dict,
        location: str,
        labor_rate: float,
        equipment_rate: float
    ):
```

```
        """Index cost estimate in vector database"""

        # Create searchable text
        text = f"""
        Materials: {', '.join(materials.keys())}
        Location: {location}
        Labor rate: {labor_rate}
        Equipment rate: {equipment_rate}
        """

        # Generate embedding
        embedding = self._generate_embedding(text)

        # Upsert to Qdrant
        self.client.upsert(
            collection_name=self.collection_name,
            points=[
                PointStruct(
                    id=int(estimate_id.replace('-', '')[:15]),  # Convert to int
                    vector=embedding,
                    payload={
                        "estimate_id": str(estimate_id),
                        "materials": materials,
                        "location": location,
                        "labor_rate": labor_rate,
                        "equipment_rate": equipment_rate,
                        "text": text
                    }
                )
            ]
        )

        logger.info(f"Indexed estimate {estimate_id} to Qdrant")

    async def search_similar_estimates(
        self,
        materials: Dict,
        location: str,
        labor_rate: float,
        equipment_rate: float,
        limit: int = 5,
        score_threshold: float = 0.7
    ) -&gt; List[Dict]:
        """Search for similar cost estimates"""

        # Create search text
        search_text = f"""
        Materials: {', '.join(materials.keys())}
        Location: {location}
        Labor rate: {labor_rate}
        Equipment rate: {equipment_rate}
        """

        # Generate embedding
        query_embedding = self._generate_embedding(search_text)
```

```
        # Search in Qdrant
        search_results = self.client.search(
            collection_name=self.collection_name,
            query_vector=query_embedding,
            limit=limit,
            score_threshold=score_threshold
        )

        results = []
        for result in search_results:
            results.append({
                "estimate_id": result.payload["estimate_id"],
                "similarity_score": result.score,
                "materials": result.payload["materials"],
                "location": result.payload["location"],
                "labor_rate": result.payload["labor_rate"],
                "equipment_rate": result.payload["equipment_rate"]
            })

        return results

    async def batch_index_estimates(
        self,
        db: Session,
        estimates: List[CostEstimate]
    ):
        """Batch index multiple estimates"""

        for estimate in estimates:
            await self.index_cost_estimate(
                estimate_id=str(estimate.id),
                materials=estimate.materials,
                location=estimate.location,
                labor_rate=estimate.labor_rate,
                equipment_rate=estimate.equipment_rate
            )

# Initialize globally
vector_service = VectorSearchService()

# Routes
@router.post("/api/cost-estimates/search-similar")
async def search_similar_costs(
    query: dict,
    tenant: Tenant = Depends(get_current_tenant)
):
    """Search for similar cost estimates"""

    results = await vector_service.search_similar_estimates(
        materials=query.get("materials", {}),
        location=query.get("location", ""),
        labor_rate=query.get("labor_rate", 0),
        equipment_rate=query.get("equipment_rate", 0),
        limit=query.get("limit", 5)
    )
```

```
    return {
        "query": query,
        "similar_estimates": results,
        "average_cost": np.mean([r.get("cost", 0) for r in results]) if results else 0,
        "confidence": np.mean([r["similarity_score"] for r in results]) if results else 0
    }

@router.post("/api/cost-estimates")
async def create_cost_estimate(
    estimate: CostEstimateRequest,
    tenant: Tenant = Depends(get_current_tenant),
    db: Session = Depends(get_db)
):
    """Create cost estimate and index in vector database"""

    # Save to database
    db_estimate = CostEstimate(
        tenant_id=tenant.id,
        materials=estimate.materials,
        location=estimate.location,
        labor_rate=estimate.labor_rate,
        equipment_rate=estimate.equipment_rate
    )
    db.add(db_estimate)
    db.commit()

    # Index in vector database
    await vector_service.index_cost_estimate(
        estimate_id=str(db_estimate.id),
        materials=estimate.materials,
        location=estimate.location,
        labor_rate=estimate.labor_rate,
        equipment_rate=estimate.equipment_rate
    )

    return {"id": db_estimate.id, "created": True}
```

5 Distributed Tracing & Advanced Monitoring

Goal

Implement end-to-end request tracing and comprehensive monitoring

OpenTelemetry Setup

File: `python-services/api/monitoring/tracing.py`

```
from opentelemetry import trace, metrics
from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.exporter.prometheus import PrometheusMetricReader
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
```

```
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.sdk.metrics.export import PeriodicExportingMetricReader
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor
from opentelemetry.instrumentation.sqlalchemy import SQLAlchemyInstrumentor
from opentelemetry.instrumentation.requests import RequestsInstrumentor
from opentelemetry.instrumentation.httpx import HTTPXClientInstrumentor
import logging
import os

logger = logging.getLogger(__name__)

def init_tracing():
    """Initialize OpenTelemetry tracing"""

    # Configure Jaeger exporter
    jaeger_exporter = JaegerExporter(
        agent_host_name=os.getenv("JAEGER_HOST", "localhost"),
        agent_port=int(os.getenv("JAEGER_PORT", "6831")),
    )

    # Set tracer provider
    trace.set_tracer_provider(TracerProvider())
    trace.get_tracer_provider().add_span_processor(
        BatchSpanProcessor(jaeger_exporter)
    )

    # Auto-instrument
    FastAPIInstrumentor.instrument_app(app)
    SQLAlchemyInstrumentor().instrument(engine=engine)
    RequestsInstrumentor().instrument()
    HTTPXClientInstrumentor().instrument()

    logger.info("OpenTelemetry tracing initialized")

# Initialize tracing
init_tracing()

# Get tracer
tracer = trace.get_tracer(__name__)

# Usage in routes
@app.post("/files/{file_id}/process")
async def process_file(file_id: str):
    """Process file with distributed tracing"""

    with tracer.start_as_current_span("process_file") as main_span:
        main_span.set_attribute("file_id", file_id)
        main_span.set_attribute("tenant_id", request.scope.get("tenant_id"))

        try:
            # Extract geometry
            with tracer.start_as_current_span("extract_geometry"):
                geometry = await extract_geometry(file_id)

            # Classify materials
            with tracer.start_as_current_span("classify_materials"):
```

```
                materials = await classify_materials(geometry)

            # Estimate costs
            with tracer.start_as_current_span("estimate_costs"):
                costs = await estimate_costs(materials)

            # Calculate carbon
            with tracer.start_as_current_span("calculate_carbon"):
                carbon = await calculate_carbon(materials)

            main_span.set_attribute("status", "success")

            return {
                "geometry": geometry,
                "materials": materials,
                "costs": costs,
                "carbon": carbon
            }

        except Exception as e:
            main_span.set_attribute("status", "error")
            main_span.set_attribute("error.type", type(e).__name__)
            main_span.set_attribute("error.message", str(e))
            raise
```

Advanced Prometheus Metrics

File: `python-services/api/monitoring/metrics.py`

```
from prometheus_client import Counter, Histogram, Gauge, Summary
import time

# Define metrics
file_processing_counter = Counter(
    'file_processing_total',
    'Total files processed',
    ['tenant_id', 'file_type', 'status']
)

processing_time_histogram = Histogram(
    'processing_time_seconds',
    'File processing time in seconds',
    ['file_type'],
    buckets=(0.1, 0.5, 1.0, 2.0, 5.0, 10.0, 30.0, 60.0, 300.0)
)

active_processes_gauge = Gauge(
    'active_processes',
    'Number of active processes',
    ['process_type']
)

api_latency_summary = Summary(
    'api_latency_seconds',
    'API request latency',
```

```
    ['endpoint', 'method', 'status']
)

database_query_time = Histogram(
    'database_query_seconds',
    'Database query execution time',
    ['query_type'],
    buckets=(0.01, 0.05, 0.1, 0.5, 1.0, 5.0)
)

error_counter = Counter(
    'errors_total',
    'Total errors',
    ['error_type', 'service']
)

storage_usage_gauge = Gauge(
    'storage_usage_bytes',
    'Storage usage in bytes',
    ['tenant_id']
)

api_quota_gauge = Gauge(
    'api_quota_remaining',
    'Remaining API quota',
    ['tenant_id']
)

# Middleware to track metrics
@app.middleware("http")
async def track_metrics(request: Request, call_next):
    start_time = time.time()

    response = await call_next(request)

    process_time = time.time() - start_time

    # Track latency
    api_latency_summary.labels(
        endpoint=request.url.path,
        method=request.method,
        status=response.status_code
    ).observe(process_time)

    return response
```

📊 Grafana Advanced Dashboard

File: `monitoring/grafana-advanced-dashboard.json`

```
{
  "dashboard": {
    "title": "Construction AI - Phase 3 Advanced Monitoring",
    "panels": [
      {
```

```
        "title": "Distributed Request Tracing",
        "type": "nodeGraph",
        "targets": [
          {
            "expr": "rate(trace_requests_total[5m])"
          }
        ]
      },
      {
        "title": "Multi-Tenant Resource Usage",
        "type": "heatmap",
        "targets": [
          {
            "expr": "storage_usage_bytes by (tenant_id)"
          }
        ]
      },
      {
        "title": "API Quota Remaining",
        "type": "stat",
        "targets": [
          {
            "expr": "api_quota_remaining by (tenant_id)"
          }
        ]
      },
      {
        "title": "Database Performance",
        "type": "graph",
        "targets": [
          {
            "expr": "histogram_quantile(0.95, database_query_seconds_bucket)"
          }
        ]
      },
      {
        "title": "Vector Search Performance",
        "type": "graph",
        "targets": [
          {
            "expr": "qdrant_search_latency_seconds"
          }
        ]
      },
      {
        "title": "Archival Status",
        "type": "stat",
        "targets": [
          {
            "expr": "archival_files_total"
          }
        ]
      }
    ]
  }
}
```

🚀 Phase 3 Deployment Steps

Step 1: Database Migrations

```
psql -U user -d construction_ai -f migrations/phase3_multi_tenancy.sql
```

Step 2: Setup Qdrant

```
docker run -d \
  --name qdrant \
  -p 6333:6333 \
  qdrant/qdrant:latest
```

Step 3: Deploy Updated Services

```
# Update dependencies
pip install qdrant-client sentence-transformers opentelemetry-api opentelemetry-sdk opent

# Deploy
docker-compose up -d api
```

Step 4: Initialize Tenants

```
python scripts/create_tenants.py
```

Step 5: Setup Monitoring

```
docker-compose up -d jaeger prometheus grafana
```

✅ Phase 3 Testing Checklist

- [ ] Tenant isolation working (can't access other tenant's data)
- [ ] Multi-tenant API requests routed correctly
- [ ] Usage metrics tracking accurately
- [ ] Billing calculations correct
- [ ] Invoice generation working
- [ ] File archival to S3 working
- [ ] File restoration from Glacier working
- [ ] Vector search finding similar estimates

- [ ] Distributed tracing visible in Jaeger
- [ ] Prometheus metrics collecting data
- [ ] Grafana dashboards displaying correctly

🎯 Success Metrics (Phase 3)

✅ **Expected Results:**

- Complete data isolation for tenants
- Accurate usage tracking and billing
- 100% file archival success rate
- Vector search accuracy > 95%
- End-to-end request tracing working
- Enterprise-ready platform

🚀 After Phase 3

When Phase 3 is complete, you're ready for:

1. **Phase 4:** Performance optimization & scaling
2. **Production Launch** with enterprise support
3. **Multi-tenant Deployments** at scale

Phase 3 Total Effort: ~160 hours
Estimated Completion: 4 weeks

Ready to implement Phase 3? Start with multi-tenancy foundation!