🎯 Prompt Mühendisliği Uygulaması - Kapsamlı Geliştirme Planı

📋 Proje Genel Bakış

Proje Adı: Prompt Mühendisliği Uygulaması
Geliştirme Süresi: 8-10 hafta
Ekip Büyüklüğü: 3-5 kişi (1 backend, 2 frontend, 1 AI/ML, 1 DevOps)
Teknoloji Stack: Node.js, React, PostgreSQL, Redis, Docker

Mevcut Durum Analizi

✅ Güçlü Yanlar:

- Modern, responsive HTML/CSS/JS uygulaması
- 7 farklı AI model desteği (Claude 4, GPT-4.1, GPT-4o, Gemini 2.5, DeepSeek R1, Llama 4, Grok 3)
- 6 prompt tekniği entegrasyonu (CoT, ToT, RAG, Constitutional, Multimodal, Meta)
- 4 hazır şablon sistemi (Akademik, Teknik, İş, Yaratıcı)
- İyi tasarım ve kullanıcı deneyimi

⚠️ Geliştirme Gereksinimleri:

- Backend altyapısı eksik
- Gerçek AI API entegrasyonu yok
- Kullanıcı yönetimi yok
- Veri persistency yok
- Analytics ve raporlama eksik
- Topluluk özellikleri yok

🏗️ FAZ 1: Backend Altyapısı (2-3 Hafta)

1.1 Teknoloji Stack Seçimi

```javascript
```

```
// Önerilen Stack
Backend: Node.js + Express.js + TypeScript
Database: PostgreSQL + Redis (cache)
ORM: Prisma
Authentication: JWT + bcrypt
Validation: Joi
Rate Limiting: express-rate-limit
Documentation: Swagger/OpenAPI
Testing: Jest + Supertest
Monitoring: Prometheus + Grafana
```

1.2 Proje Yapısı

bash

```
prompt-engineering-backend/
├── src/
│   ├── controllers/        # Route handlers
│   │   ├── authController.js
│   │   ├── promptController.js
│   │   ├── userController.js
│   │   └── analyticsController.js
│   ├── services/           # Business logic
│   │   ├── aiProviders/
│   │   │   ├── anthropicProvider.js
│   │   │   ├── openaiProvider.js
│   │   │   ├── googleProvider.js
│   │   │   └── providerManager.js
│   │   ├── promptService.js
│   │   ├── authService.js
│   │   └── analyticsService.js
│   ├── models/             # Data models
│   │   ├── User.js
│   │   ├── Prompt.js
│   │   └── Analytics.js
│   ├── middleware/         # Express middleware
│   │   ├── auth.js
│   │   ├── rateLimiter.js
│   │   ├── validator.js
│   │   └── errorHandler.js
│   ├── routes/             # API routes
│   │   ├── auth.js
│   │   ├── prompts.js
│   │   ├── users.js
│   │   └── analytics.js
│   ├── utils/              # Helper functions
│   │   ├── logger.js
│   │   ├── encryption.js
│   │   └── helpers.js
│   ├── config/             # Configuration
│   │   ├── database.js
│   │   ├── redis.js
│   │   └── environment.js
│   └── types/              # TypeScript types
│       ├── User.ts
│       ├── Prompt.ts
│       └── API.ts
├── prisma/                 # Database schema
│   ├── schema.prisma
│   └── migrations/
├── tests/                  # Test files
```

```
│   ├── unit/
│   ├── integration/
│   └── helpers/
├── docker/                 # Docker configurations
│   ├── Dockerfile
│   ├── docker-compose.yml
│   └── nginx.conf
└── docs/                   # API documentation
    ├── swagger.yml
    └── README.md
```

1.3 Veritabanı Şeması

sql

```sql
-- PostgreSQL Schema Design

-- Kullanıcılar tablosu
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    name VARCHAR(255) NOT NULL,
    avatar_url VARCHAR(500),
    subscription_tier VARCHAR(50) DEFAULT 'free',
    api_usage_limit INTEGER DEFAULT 100,
    api_usage_current INTEGER DEFAULT 0,
    api_usage_reset_date TIMESTAMP DEFAULT (NOW() + INTERVAL '1 month'),
    preferences JSONB DEFAULT '{}',
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW(),
    last_login TIMESTAMP,
    is_active BOOLEAN DEFAULT TRUE,
    email_verified BOOLEAN DEFAULT FALSE,
    two_factor_enabled BOOLEAN DEFAULT FALSE
);

-- Prompt'lar tablosu
CREATE TABLE prompts (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    title VARCHAR(255) NOT NULL,
    description TEXT,
    original_input TEXT NOT NULL,
    generated_prompt TEXT NOT NULL,
    model_used VARCHAR(100) NOT NULL,
    technique_used VARCHAR(100) NOT NULL,
    template_used VARCHAR(100),
    tags VARCHAR(255)[],
    is_public BOOLEAN DEFAULT FALSE,
    is_featured BOOLEAN DEFAULT FALSE,
    is_template BOOLEAN DEFAULT FALSE,
    version INTEGER DEFAULT 1,
    parent_prompt_id INTEGER REFERENCES prompts(id),
    fork_count INTEGER DEFAULT 0,
    view_count INTEGER DEFAULT 0,
    metadata JSONB DEFAULT '{}',
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);
```

```sql
-- Prompt performans metrikleri
CREATE TABLE prompt_analytics (
    id SERIAL PRIMARY KEY,
    prompt_id INTEGER REFERENCES prompts(id) ON DELETE CASCADE,
    user_id INTEGER REFERENCES users(id),
    usage_count INTEGER DEFAULT 0,
    success_rate DECIMAL(5,2) DEFAULT 0,
    avg_response_time INTEGER, -- milliseconds
    total_tokens_used INTEGER DEFAULT 0,
    cost_usd DECIMAL(10,4) DEFAULT 0,
    user_rating INTEGER CHECK (user_rating >= 1 AND user_rating <= 5),
    feedback TEXT,
    performance_metrics JSONB DEFAULT '{}',
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- AI API kullanım logları
CREATE TABLE api_usage_logs (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    prompt_id INTEGER REFERENCES prompts(id),
    provider VARCHAR(50) NOT NULL, -- 'openai', 'anthropic', etc.
    model VARCHAR(100) NOT NULL,
    input_tokens INTEGER,
    output_tokens INTEGER,
    cost_usd DECIMAL(10,6),
    latency_ms INTEGER,
    status VARCHAR(20), -- 'success', 'error', 'timeout'
    error_message TEXT,
    request_data JSONB,
    response_data JSONB,
    created_at TIMESTAMP DEFAULT NOW()
);

-- Topluluk özellikleri
CREATE TABLE prompt_likes (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    prompt_id INTEGER REFERENCES prompts(id) ON DELETE CASCADE,
    created_at TIMESTAMP DEFAULT NOW(),
    UNIQUE(user_id, prompt_id)
);

CREATE TABLE prompt_comments (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
```

```
    prompt_id INTEGER REFERENCES prompts(id) ON DELETE CASCADE,
    content TEXT NOT NULL,
    parent_comment_id INTEGER REFERENCES prompt_comments(id),
    is_edited BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE prompt_saves (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    prompt_id INTEGER REFERENCES prompts(id) ON DELETE CASCADE,
    created_at TIMESTAMP DEFAULT NOW(),
    UNIQUE(user_id, prompt_id)
);

CREATE TABLE prompt_shares (
    id SERIAL PRIMARY KEY,
    prompt_id INTEGER REFERENCES prompts(id) ON DELETE CASCADE,
    user_id INTEGER REFERENCES users(id),
    share_token VARCHAR(255) UNIQUE,
    expires_at TIMESTAMP,
    view_count INTEGER DEFAULT 0,
    created_at TIMESTAMP DEFAULT NOW()
);

-- Takım yönetimi (Enterprise)
CREATE TABLE teams (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    owner_id INTEGER REFERENCES users(id),
    settings JSONB DEFAULT '{}',
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE team_members (
    id SERIAL PRIMARY KEY,
    team_id INTEGER REFERENCES teams(id) ON DELETE CASCADE,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    role VARCHAR(50) DEFAULT 'member', -- 'owner', 'admin', 'member'
    permissions JSONB DEFAULT '{}',
    joined_at TIMESTAMP DEFAULT NOW(),
    UNIQUE(team_id, user_id)
);
```

```sql
-- Abonelik ve faturalandırma
CREATE TABLE subscriptions (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    plan VARCHAR(50) NOT NULL, -- 'free', 'pro', 'enterprise'
    status VARCHAR(20) DEFAULT 'active', -- 'active', 'cancelled', 'expired'
    current_period_start TIMESTAMP,
    current_period_end TIMESTAMP,
    stripe_subscription_id VARCHAR(255),
    stripe_customer_id VARCHAR(255),
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE billing_records (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    subscription_id INTEGER REFERENCES subscriptions(id),
    amount_usd DECIMAL(10,2),
    description TEXT,
    invoice_url VARCHAR(500),
    status VARCHAR(20), -- 'pending', 'paid', 'failed'
    stripe_invoice_id VARCHAR(255),
    created_at TIMESTAMP DEFAULT NOW()
);

-- Sistem metrikleri
CREATE TABLE system_metrics (
    id SERIAL PRIMARY KEY,
    metric_name VARCHAR(100) NOT NULL,
    metric_value DECIMAL(15,6),
    metric_type VARCHAR(50), -- 'counter', 'gauge', 'histogram'
    tags JSONB DEFAULT '{}',
    timestamp TIMESTAMP DEFAULT NOW()
);

-- İndeksler ve Optimizasyonlar
CREATE INDEX idx_prompts_user_id ON prompts(user_id);
CREATE INDEX idx_prompts_public ON prompts(is_public) WHERE is_public = TRUE;
CREATE INDEX idx_prompts_featured ON prompts(is_featured) WHERE is_featured = TRUE;
CREATE INDEX idx_prompts_tags ON prompts USING GIN(tags);
CREATE INDEX idx_prompts_created_at ON prompts(created_at DESC);
CREATE INDEX idx_analytics_prompt_id ON prompt_analytics(prompt_id);
CREATE INDEX idx_api_logs_user_date ON api_usage_logs(user_id, created_at DESC);
CREATE INDEX idx_api_logs_provider ON api_usage_logs(provider, created_at DESC);
CREATE INDEX idx_likes_prompt_id ON prompt_likes(prompt_id);
CREATE INDEX idx_comments_prompt_id ON prompt_comments(prompt_id, created_at DESC);
```

```
CREATE INDEX idx_saves_user_id ON prompt_saves(user_id, created_at DESC);

-- Veritabanı fonksiyonları
CREATE OR REPLACE FUNCTION update_updated_at_column()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ language 'plpgsql';

-- Trigger'lar
CREATE TRIGGER update_users_updated_at BEFORE UPDATE ON users FOR EACH ROW EXECUTE FUNCTION u
CREATE TRIGGER update_prompts_updated_at BEFORE UPDATE ON prompts FOR EACH ROW EXECUTE FUNCT
CREATE TRIGGER update_analytics_updated_at BEFORE UPDATE ON prompt_analytics FOR EACH ROW EXECUT
```

1.4 API Endpoint'leri

javascript

```
// src/routes/auth.js
const express = require('express');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const { body, validationResult } = require('express-validator');
const rateLimit = require('express-rate-limit');

const router = express.Router();

// Rate limiting for auth endpoints
const authLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 dakika
  max: 5, // max 5 attempt
  message: 'Çok fazla giriş denemesi, lütfen 15 dakika sonra tekrar deneyin'
});

// Kullanıcı kaydı
router.post('/register', authLimiter, [
  body('email').isEmail().normalizeEmail(),
  body('password').isLength({ min: 8 }).matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*
  body('name').isLength({ min: 2, max: 50 }).trim()
], async (req, res) => {
  try {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    const { email, password, name } = req.body;

    // Email kontrolü
    const existingUser = await prisma.user.findUnique({ where: { email } });
    if (existingUser) {
      return res.status(400).json({ error: 'Bu email zaten kullanımda' });
    }

    // Password hash
    const passwordHash = await bcrypt.hash(password, 12);

    // Kullanıcı oluştur
    const user = await prisma.user.create({
      data: {
        email,
        passwordHash,
        name,
        subscriptionTier: 'free',
```

```
            apiUsageLimit: 100,
            apiUsageCurrent: 0
        }
    });

    // JWT token oluştur
    const token = jwt.sign(
        { userId: user.id, email: user.email },
        process.env.JWT_SECRET,
        { expiresIn: '7d' }
    );

    // Log user registration
    logger.info('User registered', { userId: user.id, email: user.email });

    res.status(201).json({
        success: true,
        user: {
            id: user.id,
            email: user.email,
            name: user.name,
            subscriptionTier: user.subscriptionTier
        },
        token
    });
  } catch (error) {
    logger.error('Registration error:', error);
    res.status(500).json({ error: 'Sunucu hatası' });
  }
});

// Giriş
router.post('/login', authLimiter, [
  body('email').isEmail().normalizeEmail(),
  body('password').notEmpty()
], async (req, res) => {
  try {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    const { email, password } = req.body;

    // Kullanıcı kontrolü
    const user = await prisma.user.findUnique({ where: { email } });
    if (!user || !user.isActive) {
```

```
            return res.status(401).json({ error: 'Geçersiz email veya şifre' });
        }

        // Şifre kontrolü
        const isValidPassword = await bcrypt.compare(password, user.passwordHash);
        if (!isValidPassword) {
            return res.status(401).json({ error: 'Geçersiz email veya şifre' });
        }

        // Son giriş güncelle
        await prisma.user.update({
            where: { id: user.id },
            data: { lastLogin: new Date() }
        });

        // JWT token
        const token = jwt.sign(
            { userId: user.id, email: user.email },
            process.env.JWT_SECRET,
            { expiresIn: '7d' }
        );

        logger.info('User logged in', { userId: user.id, email: user.email });

        res.json({
            success: true,
            user: {
                id: user.id,
                email: user.email,
                name: user.name,
                subscriptionTier: user.subscriptionTier,
                apiUsageCurrent: user.apiUsageCurrent,
                apiUsageLimit: user.apiUsageLimit
            },
            token
        });
    } catch (error) {
        logger.error('Login error:', error);
        res.status(500).json({ error: 'Sunucu hatası' });
    }
});

// Şifre sıfırlama talebi
router.post('/forgot-password', [
    body('email').isEmail().normalizeEmail()
], async (req, res) => {
    try {
```

```
    const { email } = req.body;

    const user = await prisma.user.findUnique({ where: { email } });
    if (!user) {
      // Güvenlik için her zaman başarılı mesajı döndür
      return res.json({ success: true, message: 'Şifre sıfırlama talimatları email adresinize gönderildi' });
    }

    // Reset token oluştur
    const resetToken = crypto.randomBytes(32).toString('hex');
    const resetTokenExpiry = new Date(Date.now() + 3600000); // 1 saat

    await prisma.user.update({
      where: { id: user.id },
      data: {
        resetToken,
        resetTokenExpiry
      }
    });

    // Email gönder
    await emailService.sendPasswordResetEmail(user.email, resetToken);

    res.json({ success: true, message: 'Şifre sıfırlama talimatları email adresinize gönderildi' });
  } catch (error) {
    logger.error('Password reset error:', error);
    res.status(500).json({ error: 'Sunucu hatası' });
  }
});

module.exports = router;
```

javascript

```
// src/routes/prompts.js
const express = require('express');
const { authenticateToken } = require('../middleware/auth');
const { generatePrompt } = require('../services/promptService');
const { body, query, param, validationResult } = require('express-validator');
const rateLimit = require('express-rate-limit');

const router = express.Router();

// Rate limiting for prompt generation
const generateLimiter = rateLimit({
  windowMs: 60 * 1000, // 1 dakika
  max: 10, // max 10 prompt per minute
  keyGenerator: (req) => req.user.userId
});

// Prompt oluştur
router.post('/generate', authenticateToken, generateLimiter, [
  body('input').isLength({ min: 10, max: 5000 }).trim(),
  body('model').isIn(['claude4', 'gpt41', 'gpt4o', 'gemini25', 'deepseek', 'llama4', 'grok3']),
  body('technique').isIn(['cot', 'tot', 'rag', 'constitutional', 'multimodal', 'meta']),
  body('template').optional().isIn(['academic', 'technical', 'business', 'creative']),
  body('title').optional().isLength({ max: 255 }).trim()
], async (req, res) => {
  try {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    const { input, model, technique, template, title, isPublic = false } = req.body;
    const userId = req.user.userId;

    // Kullanım limiti kontrolü
    const user = await prisma.user.findUnique({ where: { id: userId } });
    if (user.apiUsageCurrent >= user.apiUsageLimit) {
      return res.status(429).json({
        error: 'API kullanım limitiniz doldu',
        limit: user.apiUsageLimit,
        current: user.apiUsageCurrent,
        resetDate: user.apiUsageResetDate
      });
    }

    // Maliyet tahmini
    const costEstimate = await promptService.estimateCost(input, { model, technique });
```

```
if (costEstimate > user.remainingBudget) {
  return res.status(402).json({
    error: 'Yetersiz bakiye',
    estimated: costEstimate,
    available: user.remainingBudget
  });
}

// Prompt oluştur
const result = await promptService.generatePrompt({
  input,
  model,
  technique,
  template,
  userId
});

// Veritabanına kaydet
const savedPrompt = await prisma.prompt.create({
  data: {
    userId,
    title: title || `${technique.toUpperCase()} Prompt - ${new Date().toLocaleDateString('tr-TR')}`,
    originalInput: input,
    generatedPrompt: result.prompt,
    modelUsed: model,
    techniqueUsed: technique,
    templateUsed: template,
    isPublic,
    metadata: {
      tokens: result.inputTokens + result.outputTokens,
      cost: result.cost,
      latency: result.latency,
      provider: result.provider
    }
  }
});

// Kullanım sayacını güncelle
await prisma.user.update({
  where: { id: userId },
  data: { apiUsageCurrent: user.apiUsageCurrent + 1 }
});

// Analytics kaydet
await prisma.apiUsageLog.create({
  data: {
    userId,
```

```
            promptId: savedPrompt.id,
            provider: result.provider,
            model: result.model,
            inputTokens: result.inputTokens,
            outputTokens: result.outputTokens,
            costUsd: result.cost,
            latencyMs: result.latency,
            status: 'success',
            requestData: { input, model, technique, template },
            responseData: { prompt: result.prompt }
        }
    });

    // Background analytics processing
    analyticsService.processPromptGeneration(savedPrompt.id, result);

    res.json({
        success: true,
        prompt: savedPrompt,
        usage: {
            current: user.apiUsageCurrent + 1,
            limit: user.apiUsageLimit,
            resetDate: user.apiUsageResetDate
        },
        metadata: {
            tokens: result.inputTokens + result.outputTokens,
            cost: result.cost,
            latency: result.latency,
            provider: result.provider
        }
    });
} catch (error) {
    logger.error('Prompt generation error:', error);

    // Error tracking
    await prisma.apiUsageLog.create({
        data: {
            userId: req.user.userId,
            provider: req.body.model || 'unknown',
            model: req.body.model || 'unknown',
            status: 'error',
            errorMessage: error.message,
            requestData: req.body
        }
    });

    res.status(500).json({ error: error.message || 'Prompt oluşturulurken hata oluştu' });
```

```
  }
});

// Kullanıcının promptlarını getir
router.get('/my-prompts', authenticateToken, [
  query('page').optional().isInt({ min: 1 }),
  query('limit').optional().isInt({ min: 1, max: 100 }),
  query('technique').optional().isIn(['cot', 'tot', 'rag', 'constitutional', 'multimodal', 'meta']),
  query('model').optional().isIn(['claude4', 'gpt41', 'gpt4o', 'gemini25', 'deepseek', 'llama4', 'grok3']),
  query('search').optional().isLength({ max: 255 }).trim()
], async (req, res) => {
  try {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    const userId = req.user.userId;
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 20;
    const offset = (page - 1) * limit;

    const where = { userId };
    if (req.query.technique) where.techniqueUsed = req.query.technique;
    if (req.query.model) where.modelUsed = req.query.model;
    if (req.query.search) {
      where.OR = [
        { title: { contains: req.query.search, mode: 'insensitive' } },
        { originalInput: { contains: req.query.search, mode: 'insensitive' } },
        { tags: { hasSome: [req.query.search] } }
      ];
    }

    const [prompts, total] = await Promise.all([
      prisma.prompt.findMany({
        where,
        include: {
          _count: {
            select: {
              likes: true,
              comments: true,
              saves: true
            }
          },
          analytics: {
            select: {
              usageCount: true,
```

```
                successRate: true,
                userRating: true
              },
              orderBy: { createdAt: 'desc' },
              take: 1
            }
          },
          orderBy: { createdAt: 'desc' },
          skip: offset,
          take: limit
        }),
        prisma.prompt.count({ where })
      ]);

      res.json({
        success: true,
        prompts,
        pagination: {
          page,
          limit,
          total,
          pages: Math.ceil(total / limit)
        }
      });
    } catch (error) {
      logger.error('Prompt list error:', error);
      res.status(500).json({ error: 'Sunucu hatası' });
    }
});

// Topluluk promptları
router.get('/community', [
    query('page').optional().isInt({ min: 1 }),
    query('limit').optional().isInt({ min: 1, max: 50 }),
    query('sortBy').optional().isIn(['trending', 'recent', 'popular', 'top_rated']),
    query('technique').optional().isIn(['cot', 'tot', 'rag', 'constitutional', 'multimodal', 'meta']),
    query('tag').optional().isLength({ max: 50 }).trim()
], async (req, res) => {
    try {
      const page = parseInt(req.query.page) || 1;
      const limit = parseInt(req.query.limit) || 20;
      const sortBy = req.query.sortBy || 'trending';
      const offset = (page - 1) * limit;

      const where = { isPublic: true };
      if (req.query.technique) where.techniqueUsed = req.query.technique;
      if (req.query.tag) where.tags = { has: req.query.tag };
```

```
let orderBy;
switch (sortBy) {
  case 'trending':
    orderBy = { viewCount: 'desc' };
    break;
  case 'recent':
    orderBy = { createdAt: 'desc' };
    break;
  case 'popular':
    orderBy = { forkCount: 'desc' };
    break;
  case 'top_rated':
    orderBy = { analytics: { userRating: 'desc' } };
    break;
  default:
    orderBy = { createdAt: 'desc' };
}

const [prompts, total] = await Promise.all([
  prisma.prompt.findMany({
    where,
    include: {
      user: {
        select: {
          id: true,
          name: true,
          avatarUrl: true
        }
      },
      _count: {
        select: {
          likes: true,
          comments: true,
          saves: true
        }
      }
    },
    orderBy,
    skip: offset,
    take: limit
  }),
  prisma.prompt.count({ where })
]);

res.json({
  success: true,
```

```
      prompts,
      pagination: {
        page,
        limit,
        total,
        pages: Math.ceil(total / limit)
      }
    });
  } catch (error) {
    logger.error('Community prompts error:', error);
    res.status(500).json({ error: 'Sunucu hatası' });
  }
});

// Prompt detayı
router.get('/:id', [
  param('id').isInt()
], async (req, res) => {
  try {
    const promptId = parseInt(req.params.id);

    const prompt = await prisma.prompt.findUnique({
      where: { id: promptId },
      include: {
        user: {
          select: {
            id: true,
            name: true,
            avatarUrl: true
          }
        },
        _count: {
          select: {
            likes: true,
            comments: true,
            saves: true
          }
        },
        analytics: {
          select: {
            usageCount: true,
            successRate: true,
            userRating: true
          }
        },
        comments: {
          include: {
```

```
                user: {
                  select: {
                    id: true,
                    name: true,
                    avatarUrl: true
                  }
                }
              },
              orderBy: { createdAt: 'desc' },
              take: 10
            }
          }
        });

        if (!prompt) {
          return res.status(404).json({ error: 'Prompt bulunamadı' });
        }

        // Public olmayan prompt'ları sadece sahibi görebilir
        if (!prompt.isPublic && (!req.user || req.user.userId !== prompt.userId)) {
          return res.status(403).json({ error: 'Bu prompt'a erişim yetkiniz yok' });
        }

        // View count artır
        await prisma.prompt.update({
          where: { id: promptId },
          data: { viewCount: { increment: 1 } }
        });

        res.json({
          success: true,
          prompt
        });
      } catch (error) {
        logger.error('Prompt detail error:', error);
        res.status(500).json({ error: 'Sunucu hatası' });
      }
    });

    // Prompt beğen/beğenmekten vazgeç
    router.post('/:id/like', authenticateToken, [
      param('id').isInt()
    ], async (req, res) => {
      try {
        const promptId = parseInt(req.params.id);
        const userId = req.user.userId;
```

```
        const existingLike = await prisma.promptLike.findUnique({
            where: {
                userId_promptId: {
                    userId,
                    promptId
                }
            }
        });

        if (existingLike) {
            // Unlike
            await prisma.promptLike.delete({
                where: { id: existingLike.id }
            });
        } else {
            // Like
            await prisma.promptLike.create({
                data: {
                    userId,
                    promptId
                }
            });
        }

        const likeCount = await prisma.promptLike.count({
            where: { promptId }
        });

        res.json({
            success: true,
            liked: !existingLike,
            likeCount
        });
    } catch (error) {
        logger.error('Like error:', error);
        res.status(500).json({ error: 'Sunucu hatası' });
    }
});

// Prompt fork et
router.post('/:id/fork', authenticateToken, [
    param('id').isInt(),
    body('title').optional().isLength({ max: 255 }).trim()
], async (req, res) => {
    try {
        const originalPromptId = parseInt(req.params.id);
        const userId = req.user.userId;
```

```
    const { title } = req.body;

    const originalPrompt = await prisma.prompt.findUnique({
      where: { id: originalPromptId }
    });

    if (!originalPrompt || (!originalPrompt.isPublic && originalPrompt.userId !== userId)) {
      return res.status(404).json({ error: 'Prompt bulunamadı' });
    }

    // Fork oluştur
    const forkedPrompt = await prisma.prompt.create({
      data: {
        userId,
        title: title || `Fork: ${originalPrompt.title}`,
        description: originalPrompt.description,
        originalInput: originalPrompt.originalInput,
        generatedPrompt: originalPrompt.generatedPrompt,
        modelUsed: originalPrompt.modelUsed,
        techniqueUsed: originalPrompt.techniqueUsed,
        templateUsed: originalPrompt.templateUsed,
        tags: originalPrompt.tags,
        parentPromptId: originalPromptId,
        metadata: originalPrompt.metadata
      }
    });

    // Fork sayısını artır
    await prisma.prompt.update({
      where: { id: originalPromptId },
      data: { forkCount: { increment: 1 } }
    });

    res.json({
      success: true,
      prompt: forkedPrompt
    });
  } catch (error) {
    logger.error('Fork error:', error);
    res.status(500).json({ error: 'Sunucu hatası' });
  }
});

module.exports = router;
```

1.5 AI Provider Servisleri

javascript

```
// src/services/aiProviders/anthropicProvider.js
const Anthropic = require('@anthropic-ai/sdk');
const { logger } = require('../../utils/logger');

class AnthropicProvider {
  constructor() {
    this.client = new Anthropic({
      apiKey: process.env.ANTHROPIC_API_KEY
    });

    this.models = {
      'claude-3-opus': {
        name: 'claude-3-opus-20240229',
        inputPrice: 15, // $/1M tokens
        outputPrice: 75,
        maxTokens: 4096,
        contextWindow: 200000
      },
      'claude-3-sonnet': {
        name: 'claude-3-sonnet-20240229',
        inputPrice: 3,
        outputPrice: 15,
        maxTokens: 4096,
        contextWindow: 200000
      },
      'claude-3-haiku': {
        name: 'claude-3-haiku-20240307',
        inputPrice: 0.25,
        outputPrice: 1.25,
        maxTokens: 4096,
        contextWindow: 200000
      },
      'claude-4-opus': {
        name: 'claude-4-opus-20241201',
        inputPrice: 15,
        outputPrice: 75,
        maxTokens: 8192,
        contextWindow: 500000
      }
    };

    this.rateLimiter = new Map(); // Simple rate limiting
  }

  async generate(prompt, options = {}) {
    const startTime = Date.now();
```

```
try {
  // Rate limiting check
  await this.checkRateLimit(options.userId);

  const modelConfig = this.models[options.model] || this.models['claude-3-sonnet'];

  const systemPrompt = this.buildSystemPrompt(options.technique, options.template);
  const messages = this.buildMessages(prompt, options.technique);

  logger.info('Anthropic API request', {
    model: modelConfig.name,
    technique: options.technique,
    inputLength: prompt.length
  });

  const response = await this.client.messages.create({
    model: modelConfig.name,
    max_tokens: options.maxTokens || 4000,
    temperature: options.temperature || 0.7,
    messages,
    system: systemPrompt
  });

  const latency = Date.now() - startTime;
  const inputTokens = response.usage.input_tokens;
  const outputTokens = response.usage.output_tokens;

  const cost = this.calculateCost(inputTokens, outputTokens, options.model);

  const result = {
    content: response.content[0].text,
    provider: 'anthropic',
    model: options.model,
    inputTokens,
    outputTokens,
    latency,
    cost,
    usage: response.usage,
    metadata: {
      technique: options.technique,
      template: options.template,
      timestamp: new Date().toISOString()
    }
  };

  logger.info('Anthropic API response', {
```

```
            model: modelConfig.name,
            inputTokens,
            outputTokens,
            latency,
            cost
        });

        return result;
    } catch (error) {
        logger.error('Anthropic API error:', {
            error: error.message,
            model: options.model,
            technique: options.technique
        });

        if (error.status === 429) {
            throw new Error('Rate limit aşıldı. Lütfen biraz bekleyip tekrar deneyin.');
        } else if (error.status === 401) {
            throw new Error('API anahtarı geçersiz.');
        } else if (error.status === 400) {
            throw new Error('Geçersiz istek parametreleri.');
        }

        throw new Error(`Anthropic API hatası: ${error.message}`);
    }
}

buildSystemPrompt(technique, template) {
    const baseSystems = {
        cot: 'Sen adım adım düşünen ve analitik yaklaşan bir uzmansın. Her problemi mantıklı adımlara böl ve açıkla
        tot: 'Sen çoklu yaklaşım değerlendiren ve en iyi çözümü seçen bir strateji uzmanısın.',
        rag: 'Sen verilen kaynaklara dayanarak yanıt veren ve her iddiayı kaynaklarla destekleyen bir araştırmacısın.'
        constitutional: 'Sen etik kurallara uygun, zararsız ve faydalı yanıtlar veren sorumlu bir asistansın.',
        multimodal: 'Sen farklı türdeki bilgileri entegre eden ve kapsamlı analiz yapan bir uzmansın.',
        meta: 'Sen en etkili yaklaşımı belirleyen ve kendi düşünce sürecini optimize eden bir meta-uzman\'sın.'
    };

    const templateSystems = {
        academic: 'Akademik standartlarda, kaynak belirtilerek ve bilimsel metodoloji kullanarak yanıtla.',
        technical: 'Teknik detaylarla, adım adım açıklamalarla ve kod örnekleriyle yanıtla.',
        business: 'İş dünyası perspektifiyle, ROI odaklı ve eylem planları içeren yanıtlar ver.',
        creative: 'Yaratıcı, özgün ve ilham verici yaklaşımlarla yanıtla.'
    };

    let systemPrompt = baseSystems[technique] || baseSystems.cot;

    if (template && templateSystems[template]) {
```

```
        systemPrompt += ` ${templateSystems[template]}`;
    }

    return systemPrompt;
}

buildMessages(prompt, technique) {
    const messages = [
        {
            role: 'user',
            content: prompt
        }
    ];

    return messages;
}

calculateCost(inputTokens, outputTokens, model) {
    const modelConfig = this.models[model] || this.models['claude-3-sonnet'];

    const inputCost = (inputTokens / 1000000) * modelConfig.inputPrice;
    const outputCost = (outputTokens / 1000000) * modelConfig.outputPrice;

    return Number((inputCost + outputCost).toFixed(6));
}

async estimateCost(prompt, options = {}) {
    // Rough token estimation (4 chars ≈ 1 token)
    const estimatedInputTokens = Math.ceil(prompt.length / 4);
    const estimatedOutputTokens = options.expectedOutputTokens || 1000;

    return this.calculateCost(estimatedInputTokens, estimatedOutputTokens, options.model);
}

async checkRateLimit(userId) {
    if (!userId) return;

    const key = `anthropic_${userId}`;
    const now = Date.now();
    const windowMs = 60 * 1000; // 1 dakika
    const maxRequests = 10;

    if (!this.rateLimiter.has(key)) {
        this.rateLimiter.set(key, { count: 1, resetTime: now + windowMs });
        return;
    }
```

```
        const userData = this.rateLimiter.get(key);

        if (now > userData.resetTime) {
            // Reset window
            this.rateLimiter.set(key, { count: 1, resetTime: now + windowMs });
            return;
        }

        if (userData.count >= maxRequests) {
            throw new Error('Rate limit aşıldı. Dakikada maksimum 10 istek yapabilirsiniz.');
        }

        userData.count++;
        this.rateLimiter.set(key, userData);
    }
}

module.exports = AnthropicProvider;
```

javascript

```
// src/services/aiProviders/openaiProvider.js
const OpenAI = require('openai');
const { logger } = require('../../utils/logger');

class OpenAIProvider {
  constructor() {
    this.client = new OpenAI({
      apiKey: process.env.OPENAI_API_KEY
    });

    this.models = {
      'gpt-4-turbo': {
        name: 'gpt-4-turbo-preview',
        inputPrice: 10,
        outputPrice: 30,
        maxTokens: 4096,
        contextWindow: 128000
      },
      'gpt-4': {
        name: 'gpt-4',
        inputPrice: 30,
        outputPrice: 60,
        maxTokens: 8192,
        contextWindow: 8192
      },
      'gpt-4o': {
        name: 'gpt-4o',
        inputPrice: 5,
        outputPrice: 15,
        maxTokens: 16384,
        contextWindow: 128000
      },
      'gpt-3.5-turbo': {
        name: 'gpt-3.5-turbo',
        inputPrice: 0.5,
        outputPrice: 1.5,
        maxTokens: 4096,
        contextWindow: 16385
      }
    };
  }

  async generate(prompt, options = {}) {
    const startTime = Date.now();

    try {
```

```
const modelConfig = this.models[options.model] || this.models['gpt-4-turbo'];

const messages = this.buildMessages(prompt, options.technique, options.template);

logger.info('OpenAI API request', {
  model: modelConfig.name,
  technique: options.technique,
  inputLength: prompt.length
});

const response = await this.client.chat.completions.create({
  model: modelConfig.name,
  messages,
  max_tokens: options.maxTokens || 4000,
  temperature: options.temperature || 0.7,
  response_format: options.responseFormat || undefined,
  tools: options.tools || undefined,
  tool_choice: options.toolChoice || undefined
});

const latency = Date.now() - startTime;
const inputTokens = response.usage.prompt_tokens;
const outputTokens = response.usage.completion_tokens;

const cost = this.calculateCost(inputTokens, outputTokens, options.model);

const result = {
  content: response.choices[0].message.content,
  provider: 'openai',
  model: options.model,
  inputTokens,
  outputTokens,
  latency,
  cost,
  usage: response.usage,
  metadata: {
    technique: options.technique,
    template: options.template,
    finishReason: response.choices[0].finish_reason,
    timestamp: new Date().toISOString()
  }
};

// Tool calls varsa ekle
if (response.choices[0].message.tool_calls) {
  result.toolCalls = response.choices[0].message.tool_calls;
}
```

```
      logger.info('OpenAI API response', {
        model: modelConfig.name,
        inputTokens,
        outputTokens,
        latency,
        cost,
        finishReason: response.choices[0].finish_reason
      });

      return result;
    } catch (error) {
      logger.error('OpenAI API error:', {
        error: error.message,
        model: options.model,
        technique: options.technique
      });

      if (error.status === 429) {
        throw new Error('Rate limit aşıldı. Lütfen biraz bekleyip tekrar deneyin.');
      } else if (error.status === 401) {
        throw new Error('API anahtarı geçersiz.');
      } else if (error.status === 400) {
        throw new Error('Geçersiz istek parametreleri.');
      }

      throw new Error(`OpenAI API hatası: ${error.message}`);
    }
  }

  buildMessages(prompt, technique, template) {
    const systemPrompts = {
      cot: 'Sen adım adım düşünen ve her adımı açıklayan bir problem çözücüsün. Düşünce sürecini şeffaf bir şek
      tot: 'Sen çoklu yaklaşım geliştiren ve en iyisini seçen bir stratejistsın. Farklı seçenekleri değerlendir.',
      rag: 'Sen verilen bilgilere dayanarak yanıt veren ve kaynak belirten bir araştırmacısın.',
      constitutional: 'Sen güvenli, etik ve faydalı yanıtlar veren sorumlu bir asistansın.',
      multimodal: 'Sen farklı bilgi türlerini entegre eden kapsamlı bir analist\'sin.',
      meta: 'Sen en etkili yaklaşımı belirleyen ve optimize eden bir meta-uzman\'sın.'
    };

    const templateAdditions = {
      academic: ' Akademik standartlarda, referanslarla desteklenmiş yanıtlar ver.',
      technical: ' Teknik detayları, kod örnekleri ve implementasyon adımları içer.',
      business: ' İş odaklı, ROI hesaplamalı ve aksiyon planları sun.',
      creative: ' Yaratıcı, özgün ve ilham verici çözümler öner.'
    };
```

```
        let systemPrompt = systemPrompts[technique] || systemPrompts.cot;
        if (template && templateAdditions[template]) {
            systemPrompt += templateAdditions[template];
        }

        const messages = [
            {
                role: 'system',
                content: systemPrompt
            },
            {
                role: 'user',
                content: prompt
            }
        ];

        return messages;
    }

    calculateCost(inputTokens, outputTokens, model) {
        const modelConfig = this.models[model] || this.models['gpt-4-turbo'];

        const inputCost = (inputTokens / 1000000) * modelConfig.inputPrice;
        const outputCost = (outputTokens / 1000000) * modelConfig.outputPrice;

        return Number((inputCost + outputCost).toFixed(6));
    }
}

module.exports = OpenAIProvider;
```

javascript

```
// src/services/aiProviders/providerManager.js
const AnthropicProvider = require('./anthropicProvider');
const OpenAIProvider = require('./openaiProvider');
const GoogleProvider = require('./googleProvider');
const { logger } = require('../../utils/logger');

class AIProviderManager {
  constructor() {
    this.providers = {
      anthropic: new AnthropicProvider(),
      openai: new OpenAIProvider(),
      google: new GoogleProvider()
    };

    this.modelMapping = {
      'claude4': { provider: 'anthropic', model: 'claude-3-opus' },
      'claude-sonnet': { provider: 'anthropic', model: 'claude-3-sonnet' },
      'gpt41': { provider: 'openai', model: 'gpt-4-turbo' },
      'gpt4o': { provider: 'openai', model: 'gpt-4o' },
      'gemini25': { provider: 'google', model: 'gemini-1.5-pro' }
    };
  }

  async generatePrompt(input, options) {
    const modelConfig = this.modelMapping[options.model];
    if (!modelConfig) {
      throw new Error(`Desteklenmeyen model: ${options.model}`);
    }

    const provider = this.providers[modelConfig.provider];
    if (!provider) {
      throw new Error(`Desteklenmeyen sağlayıcı: ${modelConfig.provider}`);
    }

    // Cost estimation
    const costEstimate = await provider.estimateCost(input, {
      ...options,
      model: modelConfig.model
    });

    logger.info('Generating prompt', {
      provider: modelConfig.provider,
      model: modelConfig.model,
      technique: options.technique,
      estimatedCost: costEstimate
    });
```

```
    try {
      const result = await provider.generate(input, {
        ...options,
        model: modelConfig.model
      });

      // Add provider info to result
      result.provider = modelConfig.provider;
      result.modelName = modelConfig.model;

      return result;
    } catch (error) {
      logger.error('Provider generation failed', {
        provider: modelConfig.provider,
        model: modelConfig.model,
        error: error.message
      });

      // Fallback to alternative provider if available
      if (options.allowFallback) {
        return await this.tryFallback(input, options, modelConfig.provider);
      }

      throw error;
    }
  }

  async tryFallback(input, options, failedProvider) {
    const fallbackMappings = {
      'anthropic': 'openai',
      'openai': 'anthropic',
      'google': 'anthropic'
    };

    const fallbackProvider = fallbackMappings[failedProvider];
    if (!fallbackProvider) {
      throw new Error('Fallback sağlayıcı bulunamadı');
    }

    logger.info('Trying fallback provider', {
      failed: failedProvider,
      fallback: fallbackProvider
    });

    // Use default model for fallback provider
    const fallbackModel = fallbackProvider === 'anthropic' ? 'claude-3-sonnet' : 'gpt-4-turbo';
```

```
      return await this.providers[fallbackProvider].generate(input, {
        ...options,
        model: fallbackModel
      });
    }

    async estimateCost(input, options) {
      const modelConfig = this.modelMapping[options.model];
      if (!modelConfig) {
        throw new Error(`Desteklenmeyen model: ${options.model}`);
      }

      const provider = this.providers[modelConfig.provider];
      return await provider.estimateCost(input, {
        ...options,
        model: modelConfig.model
      });
    }

    getModelInfo(modelKey) {
      const modelConfig = this.modelMapping[modelKey];
      if (!modelConfig) {
        return null;
      }

      const provider = this.providers[modelConfig.provider];
      return provider.models[modelConfig.model];
    }

    getAllModels() {
      const models = {};

      for (const [key, config] of Object.entries(this.modelMapping)) {
        const provider = this.providers[config.provider];
        const modelInfo = provider.models[config.model];

        models[key] = {
          ...modelInfo,
          provider: config.provider,
          key
        };
      }

      return models;
    }
}
```

```
module.exports = AIProviderManager;
```

1.6 Docker & Deployment

dockerfile

```
# Dockerfile
FROM node:18-alpine AS builder

# Install dependencies for node-gyp
RUN apk add --no-cache python3 make g++

WORKDIR /app

# Copy package files
COPY package*.json ./
COPY prisma ./prisma/

# Install dependencies
RUN npm ci --only=production

# Generate Prisma client
RUN npx prisma generate

FROM node:18-alpine AS production

# Create app user
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nextjs -u 1001

WORKDIR /app

# Copy built application
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/prisma ./prisma
COPY . .

# Create necessary directories
RUN mkdir -p /app/logs /app/uploads
RUN chown -R nextjs:nodejs /app

USER nextjs

EXPOSE 3000

ENV NODE_ENV=production
ENV PORT=3000

HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:3000/health || exit 1
```

```
CMD ["npm", "start"]
```

yaml

```yaml
# docker-compose.yml
version: '3.8'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - DATABASE_URL=postgresql://promptuser:${DB_PASSWORD}@db:5432/promptdb
      - REDIS_URL=redis://redis:6379
      - JWT_SECRET=${JWT_SECRET}
      - ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY}
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - GOOGLE_AI_API_KEY=${GOOGLE_AI_API_KEY}
      - AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID}
      - AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY}
      - S3_BUCKET_NAME=${S3_BUCKET_NAME}
    depends_on:
      - db
      - redis
    restart: unless-stopped
    volumes:
      - ./logs:/app/logs
      - ./uploads:/app/uploads
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s

  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: promptdb
      POSTGRES_USER: promptuser
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_INITDB_ARGS: "--encoding=UTF-8 --lc-collate=C --lc-ctype=C"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./sql/init.sql:/docker-entrypoint-initdb.d/01-init.sql
      - ./sql/seed.sql:/docker-entrypoint-initdb.d/02-seed.sql
```

```
    ports:
      - "5432:5432"
    restart: unless-stopped
    command: >
      postgres
      -c max_connections=200
      -c shared_buffers=256MB
      -c effective_cache_size=1GB
      -c maintenance_work_mem=64MB
      -c checkpoint_completion_target=0.7
      -c wal_buffers=16MB
      -c default_statistics_target=100

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    restart: unless-stopped
    command: >
      redis-server
      --appendonly yes
      --appendfsync everysec
      --maxmemory 256mb
      --maxmemory-policy allkeys-lru

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - ./nginx/conf.d:/etc/nginx/conf.d:ro
      - ./ssl:/etc/ssl/certs:ro
      - nginx_cache:/var/cache/nginx
    depends_on:
      - app
    restart: unless-stopped

  prometheus:
    image: prom/prometheus:latest
    ports:
      - "9090:9090"
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml:ro
```

```
    - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--web.enable-lifecycle'
    restart: unless-stopped

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3001:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_PASSWORD}
    volumes:
      - grafana_data:/var/lib/grafana
      - ./monitoring/grafana/dashboards:/etc/grafana/provisioning/dashboards:ro
      - ./monitoring/grafana/datasources:/etc/grafana/provisioning/datasources:ro
    restart: unless-stopped

volumes:
  postgres_data:
  redis_data:
  prometheus_data:
  grafana_data:
  nginx_cache:
```

nginx

```
# nginx/nginx.conf
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log notice;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
    use epoll;
    multi_accept on;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    # Logging
    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for" '
                    'rt=$request_time uct="$upstream_connect_time" '
                    'uht="$upstream_header_time" urt="$upstream_response_time"';

    access_log /var/log/nginx/access.log main;

    # Performance
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    types_hash_max_size 2048;
    client_max_body_size 10M;

    # Gzip compression
    gzip on;
    gzip_vary on;
    gzip_min_length 10240;
    gzip_proxied expired no-cache no-store private must-revalidate max-age=0;
    gzip_types
        text/plain
        text/css
        text/xml
        text/javascript
        application/javascript
        application/xml+rss
        application/json;
```

```
# Rate limiting
limit_req_zone $binary_remote_addr zone=api:10m rate=10r/s;
limit_req_zone $binary_remote_addr zone=auth:10m rate=5r/m;

# Upstream backend
upstream app_backend {
    server app:3000;
    keepalive 32;
}

# SSL configuration
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-SHA512:ECDHE-RSA-AES256-GCM
ssl_prefer_server_ciphers off;
ssl_session_cache shared:SSL:10m;
ssl_session_timeout 10m;

# Security headers
add_header X-Frame-Options "SAMEORIGIN" always;
add_header X-Content-Type-Options "nosniff" always;
add_header X-XSS-Protection "1; mode=block" always;
add_header Referrer-Policy "no-referrer-when-downgrade" always;
add_header Content-Security-Policy "default-src 'self' http: https: data: blob: 'unsafe-inline'" always;

server {
    listen 80;
    server_name _;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl http2;
    server_name _;

    ssl_certificate /etc/ssl/certs/fullchain.pem;
    ssl_certificate_key /etc/ssl/certs/privkey.pem;

    # API routes
    location /api/ {
        limit_req zone=api burst=20 nodelay;

        proxy_pass http://app_backend;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
```

```
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_cache_bypass $http_upgrade;

    # Timeouts
    proxy_connect_timeout 5s;
    proxy_send_timeout 60s;
    proxy_read_timeout 60s;
}

# Auth routes (stricter rate limiting)
location /api/auth/ {
    limit_req zone=auth burst=5 nodelay;

    proxy_pass http://app_backend;
    proxy_http_version 1.1;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}

# Static files
location /static/ {
    expires 1y;
    add_header Cache-Control "public, immutable";
    proxy_pass http://app_backend;
}

# Health check
location /health {
    proxy_pass http://app_backend;
    access_log off;
}

# Metrics (internal only)
location /metrics {
    allow 127.0.0.1;
    allow 10.0.0.0/8;
    allow 172.16.0.0/12;
    allow 192.168.0.0/16;
    deny all;

    proxy_pass http://app_backend;
}
```

```
        # Frontend (React app)
        location / {
            proxy_pass http://app_backend;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
        }
    }
}
```

⚛️ FAZ 2: Frontend Modernizasyonu (2 Hafta)

2.1 React'e Geçiş ve Proje Kurulumu

bash

```
# Yeni proje oluştur
npx create-react-app prompt-engineering-frontend --template typescript
cd prompt-engineering-frontend

# Gerekli kütüphaneleri yükle
npm install @tanstack/react-query zustand react-router-dom
npm install @headlessui/react @heroicons/react
npm install tailwindcss @tailwindcss/forms @tailwindcss/typography autoprefixer postcss
npm install react-hook-form @hookform/resolvers yup
npm install axios react-hot-toast react-markdown
npm install @codemirror/react-codemirror @codemirror/lang-markdown
npm install framer-motion lucide-react
npm install recharts chart.js react-chartjs-2
npm install react-virtualized-auto-sizer react-window

# Development dependencies
npm install --save-dev @types/react @types/react-dom
npm install --save-dev @testing-library/react @testing-library/jest-dom @testing-library/user-event
npm install --save-dev cypress @cypress/react @cypress/code-coverage
npm install --save-dev prettier eslint-config-prettier
npm install --save-dev husky lint-staged
```

2.2 Proje Yapısı

bash

```
src/
├── components/          # React components
│   ├── ui/              # Reusable UI components
│   │   ├── Button.tsx
│   │   ├── Input.tsx
│   │   ├── Modal.tsx
│   │   ├── LoadingSpinner.tsx
│   │   └── index.ts
│   ├── layout/          # Layout components
│   │   ├── Header.tsx
│   │   ├── Sidebar.tsx
│   │   ├── Footer.tsx
│   │   └── MainLayout.tsx
│   ├── PromptGenerator/ # Prompt generation components
│   │   ├── PromptGenerator.tsx
│   │   ├── ModelSelector.tsx
│   │   ├── TechniqueSelector.tsx
│   │   ├── PromptInput.tsx
│   │   ├── PromptOutput.tsx
│   │   └── index.ts
│   ├── Community/       # Community features
│   │   ├── CommunityHub.tsx
│   │   ├── PromptCard.tsx
│   │   ├── SearchFilters.tsx
│   │   └── index.ts
│   ├── Analytics/       # Analytics components
│   │   ├── AnalyticsDashboard.tsx
│   │   ├── MetricCard.tsx
│   │   ├── Charts/
│   │   │   ├── UsageChart.tsx
│   │   │   ├── CostChart.tsx
│   │   │   └── ModelChart.tsx
│   │   └── index.ts
│   └── Auth/            # Authentication components
│       ├── LoginForm.tsx
│       ├── RegisterForm.tsx
│       ├── ProtectedRoute.tsx
│       └── index.ts
├── hooks/               # Custom React hooks
│   ├── useAuth.ts
│   ├── usePromptGeneration.ts
│   ├── useDebounce.ts
│   ├── useLocalStorage.ts
│   └── useAnalytics.ts
├── services/            # API services
│   ├── api.ts
```

```
│   ├── authService.ts
│   ├── promptService.ts
│   ├── communityService.ts
│   └── analyticsService.ts
├── stores/          # State management
│   ├── authStore.ts
│   ├── promptStore.ts
│   ├── uiStore.ts
│   └── index.ts
├── types/           # TypeScript types
│   ├── api.ts
│   ├── prompt.ts
│   ├── user.ts
│   └── index.ts
├── utils/           # Utility functions
│   ├── constants.ts
│   ├── formatters.ts
│   ├── validators.ts
│   └── helpers.ts
├── styles/          # Styling
│   ├── globals.css
│   ├── components.css
│   └── tailwind.css
└── pages/           # Page components
    ├── HomePage.tsx
    ├── GeneratorPage.tsx
    ├── CommunityPage.tsx
    ├── AnalyticsPage.tsx
    ├── ProfilePage.tsx
    └── index.ts
```

2.3 Modern Component Mimarisi

tsx

```tsx
// src/components/PromptGenerator/PromptGenerator.tsx
import React, { useState, useCallback } from 'react';
import { motion, AnimatePresence } from 'framer-motion';
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query';
import { toast } from 'react-hot-toast';
import { usePromptStore } from '../../stores/promptStore';
import { useAuthStore } from '../../stores/authStore';
import { promptService } from '../../services/promptService';
import { ModelSelector } from './ModelSelector';
import { TechniqueSelector } from './TechniqueSelector';
import { PromptInput } from './PromptInput';
import { PromptOutput } from './PromptOutput';
import { LoadingSpinner } from '../ui/LoadingSpinner';
import { Button } from '../ui/Button';
import { Card } from '../ui/Card';
import { UsageIndicator } from './UsageIndicator';
import type { GeneratePromptParams, PromptResult } from '../../types/prompt';

interface PromptGeneratorProps {
  className?: string;
}

export const PromptGenerator: React.FC<PromptGeneratorProps> = ({ className }) => {
  const [formData, setFormData] = useState<GeneratePromptParams>({
    input: '',
    model: 'claude4',
    technique: 'cot',
    template: '',
    title: '',
    isPublic: false
  });

  const queryClient = useQueryClient();
  const { user } = useAuthStore();
  const { addPrompt, recentPrompts } = usePromptStore();

  // Kullanıcı bilgilerini getir
  const { data: userStats } = useQuery({
    queryKey: ['user-stats'],
    queryFn: () => promptService.getUserStats(),
    enabled: !!user
  });

  // Model bilgilerini getir
  const { data: modelInfo } = useQuery({
    queryKey: ['model-info', formData.model],
```

```
  queryFn: () => promptService.getModelInfo(formData.model)
});

// Prompt oluşturma mutation
const generateMutation = useMutation({
  mutationFn: promptService.generatePrompt,
  onSuccess: (data: PromptResult) => {
    addPrompt(data.prompt);
    toast.success('Prompt başarıyla oluşturuldu!', {
      duration: 4000,
      icon: '✨'
    });
    queryClient.invalidateQueries({ queryKey: ['user-stats'] });
  },
  onError: (error: any) => {
    const message = error.message || 'Prompt oluşturulurken hata oluştu';
    toast.error(message, {
      duration: 6000,
      icon: '❌'
    });
  }
});

// Maliyet tahmini
const { data: costEstimate } = useQuery({
  queryKey: ['cost-estimate', formData.input, formData.model],
  queryFn: () => promptService.estimateCost(formData.input, { model: formData.model }),
  enabled: formData.input.length > 10,
  staleTime: 30000 // 30 saniye
});

const handleInputChange = useCallback((field: keyof GeneratePromptParams, value: any) => {
  setFormData(prev => ({ ...prev, [field]: value }));
}, []);

const handleGenerate = async () => {
  if (!formData.input.trim()) {
    toast.error('Lütfen görevinizi tanımlayın');
    return;
  }

  if (formData.input.length < 10) {
    toast.error('Görev açıklaması en az 10 karakter olmalıdır');
    return;
  }

  await generateMutation.mutateAsync(formData);
```

```
};

const canGenerate = formData.input.trim().length >= 10 &&
                    !generateMutation.isLoading &&
                    (!userStats || userStats.apiUsageCurrent < userStats.apiUsageLimit);

return (
  <div className={`max-w-7xl mx-auto space-y-6 ${className}`}>
    {/* Kullanım durumu */}
    {userStats && (
      <motion.div
        initial={{ opacity: 0, y: -20 }}
        animate={{ opacity: 1, y: 0 }}
        transition={{ duration: 0.3 }}
      >
        <UsageIndicator
          current={userStats.apiUsageCurrent}
          limit={userStats.apiUsageLimit}
          resetDate={userStats.apiUsageResetDate}
        />
      </motion.div>
    )}

    <div className="grid grid-cols-1 lg:grid-cols-4 gap-6">
      {/* Sol panel - Ayarlar */}
      <motion.div
        className="lg:col-span-1 space-y-6"
        initial={{ opacity: 0, x: -20 }}
        animate={{ opacity: 1, x: 0 }}
        transition={{ duration: 0.3, delay: 0.1 }}
      >
        <Card>
          <h3 className="text-lg font-semibold mb-4">AI Modeli</h3>
          <ModelSelector
            value={formData.model}
            onChange={(model) => handleInputChange('model', model)}
            modelInfo={modelInfo}
          />
        </Card>

        <Card>
          <h3 className="text-lg font-semibold mb-4">Teknik</h3>
          <TechniqueSelector
            value={formData.technique}
            onChange={(technique) => handleInputChange('technique', technique)}
          />
        </Card>
```

```
        {/* Maliyet tahmini */}
        {costEstimate && (
          <Card className="bg-blue-50 border-blue-200">
            <h4 className="font-medium text-blue-900 mb-2">Maliyet Tahmini</h4>
            <div className="space-y-1 text-sm text-blue-700">
              <div className="flex justify-between">
                <span>Token:</span>
                <span>~{costEstimate.estimatedTokens}</span>
              </div>
              <div className="flex justify-between">
                <span>Maliyet:</span>
                <span>${costEstimate.estimatedCost.toFixed(4)}</span>
              </div>
            </div>
          </Card>
        )}

        {/* Son promptlar */}
        {recentPrompts.length > 0 && (
          <Card>
            <h4 className="font-medium mb-3">Son Promptlar</h4>
            <div className="space-y-2">
              {recentPrompts.slice(0, 3).map((prompt) => (
                <button
                  key={prompt.id}
                  onClick={() => handleInputChange('input', prompt.originalInput)}
                  className="w-full text-left p-2 text-sm bg-gray-50 hover:bg-gray-100 rounded transition-colors"
                >
                  <div className="font-medium truncate">{prompt.title}</div>
                  <div className="text-gray-500 text-xs">
                    {prompt.techniqueUsed.toUpperCase()}
                  </div>
                </button>
              ))}
            </div>
          </Card>
        )}
      </motion.div>

      {/* Ana alan */}
      <motion.div
        className="lg:col-span-3 space-y-6"
        initial={{ opacity: 0, x: 20 }}
        animate={{ opacity: 1, x: 0 }}
        transition={{ duration: 0.3, delay: 0.2 }}
      >
```

```
<Card>
 <PromptInput
  value={formData.input}
  title={formData.title}
  onInputChange={(input) => handleInputChange('input', input)}
  onTitleChange={(title) => handleInputChange('title', title)}
  placeholder="Görevinizi detaylı bir şekilde açıklayın..."
  maxLength={5000}
 />

 <div className="flex items-center justify-between mt-6">
  <div className="flex items-center space-x-4">
   <label className="flex items-center space-x-2">
    <input
     type="checkbox"
     checked={formData.isPublic}
     onChange={(e) => handleInputChange('isPublic', e.target.checked)}
     className="rounded border-gray-300 text-blue-600 focus:ring-blue-500"
    />
    <span className="text-sm text-gray-700">Topluluğa paylaş</span>
   </label>
  </div>

  <div className="flex gap-3">
   <Button
    variant="outline"
    onClick={() => setFormData({
     input: '',
     model: 'claude4',
     technique: 'cot',
     template: '',
     title: '',
     isPublic: false
    })}
    disabled={generateMutation.isLoading}
   >
    Temizle
   </Button>

   <Button
    onClick={handleGenerate}
    disabled={!canGenerate}
    loading={generateMutation.isLoading}
    className="min-w-[140px]"
   >
    {generateMutation.isLoading ? (
     <div className="flex items-center gap-2">
```

```
                <LoadingSpinner size="sm" />
                Oluşturuluyor...
              </div>
            ) : (
              '✨ Prompt Oluştur'
            )}
          </Button>
        </div>
      </div>
    </Card>

    {/* Sonuç alanı */}
    <AnimatePresence>
      {generateMutation.data && (
        <motion.div
          initial={{ opacity: 0, y: 20 }}
          animate={{ opacity: 1, y: 0 }}
          exit={{ opacity: 0, y: -20 }}
          transition={{ duration: 0.3 }}
        >
          <PromptOutput
            prompt={generateMutation.data.prompt}
            metadata={generateMutation.data.metadata}
          />
        </motion.div>
      )}
    </AnimatePresence>
  </motion.div>
 </div>
</div>
);
};
```

tsx

```tsx
// src/components/ui/Button.tsx
import React from 'react';
import { motion } from 'framer-motion';
import { LoadingSpinner } from './LoadingSpinner';

interface ButtonProps extends React.ButtonHTMLAttributes<HTMLButtonElement> {
  variant?: 'primary' | 'secondary' | 'outline' | 'ghost' | 'danger';
  size?: 'sm' | 'md' | 'lg';
  loading?: boolean;
  leftIcon?: React.ReactNode;
  rightIcon?: React.ReactNode;
}

const variantClasses = {
  primary: 'bg-gradient-to-r from-blue-600 to-purple-600 hover:from-blue-700 hover:to-purple-700 text-white s
  secondary: 'bg-gray-600 hover:bg-gray-700 text-white',
  outline: 'border-2 border-gray-300 hover:border-gray-400 bg-white hover:bg-gray-50 text-gray-700',
  ghost: 'hover:bg-gray-100 text-gray-700',
  danger: 'bg-red-600 hover:bg-red-700 text-white'
};

const sizeClasses = {
  sm: 'px-3 py-1.5 text-sm',
  md: 'px-4 py-2 text-base',
  lg: 'px-6 py-3 text-lg'
};

export const Button: React.FC<ButtonProps> = ({
  children,
  variant = 'primary',
  size = 'md',
  loading = false,
  leftIcon,
  rightIcon,
  disabled,
  className = '',
  ...props
}) => {
  const baseClasses = 'inline-flex items-center justify-center rounded-lg font-semibold transition-all duration-200

  return (
    <motion.button
      whileHover={!disabled && !loading ? { scale: 1.02 } : undefined}
      whileTap={!disabled && !loading ? { scale: 0.98 } : undefined}
      className={`${baseClasses} ${variantClasses[variant]} ${sizeClasses[size]} ${className}`}
      disabled={disabled || loading}
```

```
      {...props}
    >
      {loading && <LoadingSpinner size="sm" className="mr-2" />}
      {leftIcon && !loading && <span className="mr-2">{leftIcon}</span>}
      {children}
      {rightIcon && !loading && <span className="ml-2">{rightIcon}</span>}
    </motion.button>
  );
};
```

2.4 State Management (Zustand)

typescript

```typescript
// src/stores/authStore.ts
import { create } from 'zustand';
import { persist } from 'zustand/middleware';
import { authService } from '../services/authService';

interface User {
  id: string;
  email: string;
  name: string;
  avatarUrl?: string;
  subscriptionTier: string;
  apiUsageCurrent: number;
  apiUsageLimit: number;
  apiUsageResetDate: string;
}

interface AuthState {
  // State
  user: User | null;
  token: string | null;
  isLoading: boolean;
  error: string | null;

  // Actions
  login: (credentials: LoginCredentials) => Promise<void>;
  register: (data: RegisterData) => Promise<void>;
  logout: () => void;
  refreshUser: () => Promise<void>;
  clearError: () => void;
  setLoading: (loading: boolean) => void;
}

interface LoginCredentials {
  email: string;
  password: string;
}

interface RegisterData {
  email: string;
  password: string;
  name: string;
}

export const useAuthStore = create<AuthState>()(
  persist(
    (set, get) => ({
```

```
// Initial State
user: null,
token: null,
isLoading: false,
error: null,

// Actions
login: async (credentials: LoginCredentials) => {
  set({ isLoading: true, error: null });

  try {
    const response = await authService.login(credentials);

    set({
      user: response.user,
      token: response.token,
      isLoading: false
    });

    // Set token for future requests
    authService.setAuthToken(response.token);

  } catch (error: any) {
    set({
      error: error.message || 'Giriş yapılamadı',
      isLoading: false
    });
    throw error;
  }
},

register: async (data: RegisterData) => {
  set({ isLoading: true, error: null });

  try {
    const response = await authService.register(data);

    set({
      user: response.user,
      token: response.token,
      isLoading: false
    });

    authService.setAuthToken(response.token);

  } catch (error: any) {
    set({
```

```
          error: error.message || 'Kayıt olunamadı',
          isLoading: false
        });
        throw error;
      }
    },

    logout: () => {
      set({
        user: null,
        token: null,
        error: null
      });

      authService.removeAuthToken();

      // Clear other stores
      usePromptStore.getState().clearUserData();
    },

    refreshUser: async () => {
      const { token } = get();
      if (!token) return;

      try {
        const user = await authService.getCurrentUser();
        set({ user });
      } catch (error) {
        // Token might be expired, logout user
        get().logout();
      }
    },

    clearError: () => set({ error: null }),
    setLoading: (loading: boolean) => set({ isLoading: loading })
  }),
  {
    name: 'auth-storage',
    partialize: (state) => ({
      user: state.user,
      token: state.token
    })
  }
 )
);

// Initialize auth token on app start
```

```
const token = useAuthStore.getState().token;
if (token) {
  authService.setAuthToken(token);
}
```

typescript

```typescript
// src/stores/promptStore.ts
import { create } from 'zustand';
import { persist } from 'zustand/middleware';

interface Prompt {
  id: string;
  title: string;
  originalInput: string;
  generatedPrompt: string;
  modelUsed: string;
  techniqueUsed: string;
  templateUsed?: string;
  tags?: string[];
  isPublic: boolean;
  createdAt: string;
  metadata: {
    tokens: number;
    cost: number;
    latency: number;
    provider: string;
  };
}

interface PromptState {
  // State
  prompts: Prompt[];
  currentPrompt: Prompt | null;
  favorites: string[];
  recentPrompts: Prompt[];
  searchHistory: string[];

  // Filters
  filters: {
    technique: string;
    model: string;
    tag: string;
    dateRange: string;
  };

  // UI State
  isLoading: boolean;
  error: string | null;

  // Actions
  setPrompts: (prompts: Prompt[]) => void;
  addPrompt: (prompt: Prompt) => void;
```

```
  updatePrompt: (id: string, updates: Partial<Prompt>) => void;
  deletePrompt: (id: string) => void;
  setCurrentPrompt: (prompt: Prompt | null) => void;
  toggleFavorite: (promptId: string) => void;
  addToSearchHistory: (query: string) => void;
  setFilters: (filters: Partial<PromptState['filters']>) => void;
  clearFilters: () => void;
  setLoading: (loading: boolean) => void;
  setError: (error: string | null) => void;
  clearUserData: () => void;

  // Computed
  getFilteredPrompts: () => Prompt[];
  getFavoritePrompts: () => Prompt[];
  getPromptsByTechnique: (technique: string) => Prompt[];
}

export const usePromptStore = create<PromptState>()(
  persist(
    (set, get) => ({
      // Initial State
      prompts: [],
      currentPrompt: null,
      favorites: [],
      recentPrompts: [],
      searchHistory: [],
      filters: {
        technique: '',
        model: '',
        tag: '',
        dateRange: 'all'
      },
      isLoading: false,
      error: null,

      // Actions
      setPrompts: (prompts) => set({ prompts }),

      addPrompt: (prompt) => set((state) => {
        const newPrompts = [prompt, ...state.prompts];
        const newRecentPrompts = [prompt, ...state.recentPrompts.slice(0, 9)]; // Keep last 10

        return {
          prompts: newPrompts,
          recentPrompts: newRecentPrompts,
          currentPrompt: prompt
        };
```

```
}),

updatePrompt: (id, updates) => set((state) => ({
  prompts: state.prompts.map(p =>
    p.id === id ? { ...p, ...updates } : p
  ),
  currentPrompt: state.currentPrompt?.id === id
    ? { ...state.currentPrompt, ...updates }
    : state.currentPrompt
})),

deletePrompt: (id) => set((state) => ({
  prompts: state.prompts.filter(p => p.id !== id),
  favorites: state.favorites.filter(fId => fId !== id),
  recentPrompts: state.recentPrompts.filter(p => p.id !== id),
  currentPrompt: state.currentPrompt?.id === id ? null : state.currentPrompt
})),

setCurrentPrompt: (prompt) => set({ currentPrompt: prompt }),

toggleFavorite: (promptId) => set((state) => ({
  favorites: state.favorites.includes(promptId)
    ? state.favorites.filter(id => id !== promptId)
    : [...state.favorites, promptId]
})),

addToSearchHistory: (query) => set((state) => ({
  searchHistory: [
    query,
    ...state.searchHistory.filter(q => q !== query).slice(0, 9)
  ]
})),

setFilters: (newFilters) => set((state) => ({
  filters: { ...state.filters, ...newFilters }
})),

clearFilters: () => set({
  filters: {
    technique: '',
    model: '',
    tag: '',
    dateRange: 'all'
  }
}),

setLoading: (isLoading) => set({ isLoading }),
```

```
  setError: (error) => set({ error }),

  clearUserData: () => set({
    prompts: [],
    currentPrompt: null,
    favorites: [],
    recentPrompts: []
  }),

  // Computed
  getFilteredPrompts: () => {
    const { prompts, filters } = get();

    return prompts.filter(prompt => {
      if (filters.technique && prompt.techniqueUsed !== filters.technique) {
        return false;
      }
      if (filters.model && prompt.modelUsed !== filters.model) {
        return false;
      }
      if (filters.tag && !prompt.tags?.includes(filters.tag)) {
        return false;
      }
      if (filters.dateRange !== 'all') {
        const promptDate = new Date(prompt.createdAt);
        const now = new Date();
        const daysDiff = Math.floor((now.getTime() - promptDate.getTime()) / (1000 * 60 * 60 * 24));

        switch (filters.dateRange) {
          case 'today':
            if (daysDiff > 0) return false;
            break;
          case 'week':
            if (daysDiff > 7) return false;
            break;
          case 'month':
            if (daysDiff > 30) return false;
            break;
        }
      }

      return true;
    });
  },

  getFavoritePrompts: () => {
    const { prompts, favorites } = get();
```

```
      return prompts.filter(prompt => favorites.includes(prompt.id));
    },

    getPromptsByTechnique: (technique) => {
      const { prompts } = get();
      return prompts.filter(prompt => prompt.techniqueUsed === technique);
    }
  }),
  {
    name: 'prompt-storage',
    partialize: (state) => ({
      favorites: state.favorites,
      recentPrompts: state.recentPrompts,
      searchHistory: state.searchHistory,
      filters: state.filters
    })
  }
 )
);
```

2.5 Custom Hooks

typescript

```ts
// src/hooks/usePromptGeneration.ts
import { useState } from 'react';
import { useMutation, useQueryClient } from '@tanstack/react-query';
import { usePromptStore } from '../stores/promptStore';
import { promptService } from '../services/promptService';

export interface GeneratePromptParams {
  input: string;
  model: string;
  technique: string;
  template?: string;
  title?: string;
  isPublic?: boolean;
}

export const usePromptGeneration = () => {
  const queryClient = useQueryClient();
  const addPrompt = usePromptStore((state) => state.addPrompt);

  const mutation = useMutation({
    mutationFn: (params: GeneratePromptParams) =>
      promptService.generatePrompt(params),

    onSuccess: (data) => {
      addPrompt(data.prompt);
      // Kullanıcı verilerini güncelle (usage limit için)
      queryClient.invalidateQueries({ queryKey: ['user-stats'] });
      queryClient.invalidateQueries({ queryKey: ['user'] });
    }
  });

  return {
    generatePrompt: mutation.mutateAsync,
    isLoading: mutation.isLoading,
    error: mutation.error?.message || null,
    data: mutation.data,
    reset: mutation.reset
  };
};

// src/hooks/useDebounce.ts
import { useState, useEffect } from 'react';

export const useDebounce = <T>(value: T, delay: number): T => {
  const [debouncedValue, setDebouncedValue] = useState<T>(value);
```

```
  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
};

// src/hooks/useLocalStorage.ts
import { useState, useEffect } from 'react';

export const useLocalStorage = <T>(
  key: string,
  initialValue: T
): [T, (value: T | ((val: T) => T)) => void] => {
  const [storedValue, setStoredValue] = useState<T>(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.error(`Error reading localStorage key "${key}":`, error);
      return initialValue;
    }
  });

  const setValue = (value: T | ((val: T) => T)) => {
    try {
      const valueToStore = value instanceof Function ? value(storedValue) : value;
      setStoredValue(valueToStore);
      window.localStorage.setItem(key, JSON.stringify(valueToStore));
    } catch (error) {
      console.error(`Error setting localStorage key "${key}":`, error);
    }
  };

  return [storedValue, setValue];
};

// src/hooks/useIntersectionObserver.ts
import { useEffect, useRef, useState } from 'react';

export const useIntersectionObserver = (
```

```
  options: IntersectionObserverInit = {}
) => {
  const [isIntersecting, setIsIntersecting] = useState(false);
  const targetRef = useRef<HTMLDivElement>(null);

  useEffect(() => {
    const target = targetRef.current;
    if (!target) return;

    const observer = new IntersectionObserver(([entry]) => {
      setIsIntersecting(entry.isIntersecting);
    }, options);

    observer.observe(target);

    return () => {
      observer.unobserve(target);
    };
  }, [options]);

  return { targetRef, isIntersecting };
};

// src/hooks/useVirtualization.ts
import { useMemo } from 'react';
import { FixedSizeList as List } from 'react-window';

export const useVirtualization = <T>(
  items: T[],
  itemHeight: number,
  containerHeight: number
) => {
  const listProps = useMemo(() => ({
    height: containerHeight,
    itemCount: items.length,
    itemSize: itemHeight,
    itemData: items
  }), [items, itemHeight, containerHeight]);

  return { List, listProps };
};
```

2.6 PWA Özellikleri

```
json
```

```
// public/manifest.json
{
  "name": "Prompt Mühendisliği Uygulaması",
  "short_name": "PromptEngineer",
  "description": "AI Modelleri için Optimize Edilmiş Promptlar Oluşturun",
  "start_url": "/",
  "display": "standalone",
  "background_color": "#667eea",
  "theme_color": "#764ba2",
  "orientation": "portrait-primary",
  "categories": ["productivity", "developer", "ai"],
  "lang": "tr",
  "dir": "ltr",
  "scope": "/",
  "icons": [
    {
      "src": "/icons/icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png",
      "purpose": "maskable any"
    },
    {
      "src": "/icons/icon-96x96.png",
      "sizes": "96x96",
      "type": "image/png",
      "purpose": "maskable any"
    },
    {
      "src": "/icons/icon-128x128.png",
      "sizes": "128x128",
      "type": "image/png",
      "purpose": "maskable any"
    },
    {
      "src": "/icons/icon-144x144.png",
      "sizes": "144x144",
      "type": "image/png",
      "purpose": "maskable any"
    },
    {
      "src": "/icons/icon-152x152.png",
      "sizes": "152x152",
      "type": "image/png",
      "purpose": "maskable any"
    },
    {
```

```
    "src": "/icons/icon-192x192.png",
    "sizes": "192x192",
    "type": "image/png",
    "purpose": "maskable any"
   },
   {
    "src": "/icons/icon-384x384.png",
    "sizes": "384x384",
    "type": "image/png",
    "purpose": "maskable any"
   },
   {
    "src": "/icons/icon-512x512.png",
    "sizes": "512x512",
    "type": "image/png",
    "purpose": "maskable any"
   }
  ],
  "screenshots": [
   {
    "src": "/screenshots/desktop-1.png",
    "sizes": "1280x720",
    "type": "image/png",
    "form_factor": "wide",
    "label": "Ana Sayfa - Desktop"
   },
   {
    "src": "/screenshots/mobile-1.png",
    "sizes": "390x844",
    "type": "image/png",
    "form_factor": "narrow",
    "label": "Ana Sayfa - Mobil"
   }
  ],
  "prefer_related_applications": false,
  "related_applications": [],
  "shortcuts": [
   {
    "name": "Yeni Prompt",
    "short_name": "Oluştur",
    "description": "Yeni prompt oluştur",
    "url": "/generator",
    "icons": [{ "src": "/icons/shortcut-new.png", "sizes": "96x96" }]
   },
   {
    "name": "Topluluk",
    "short_name": "Keşfet",
```

```
    "description": "Topluluk prompt'larını keşfet",
    "url": "/community",
    "icons": [{ "src": "/icons/shortcut-community.png", "sizes": "96x96" }]
  }
 ]
}
```

javascript

```
// public/sw.js - Service Worker
const CACHE_NAME = 'prompt-engineering-v2.1.0';
const STATIC_CACHE_NAME = 'static-v2.1.0';
const DYNAMIC_CACHE_NAME = 'dynamic-v2.1.0';

const STATIC_ASSETS = [
  '/',
  '/static/css/main.css',
  '/static/js/main.js',
  '/manifest.json',
  '/icons/icon-192x192.png',
  '/icons/icon-512x512.png',
  '/offline.html'
];

const DYNAMIC_CACHE_LIMIT = 50;

// Install event
self.addEventListener('install', (event) => {
  console.log('Service Worker installing...');

  event.waitUntil(
    caches.open(STATIC_CACHE_NAME)
      .then((cache) => {
        console.log('Caching static assets');
        return cache.addAll(STATIC_ASSETS);
      })
      .then(() => {
        console.log('Service Worker installed');
        return self.skipWaiting();
      })
  );
});

// Activate event
self.addEventListener('activate', (event) => {
  console.log('Service Worker activating...');

  event.waitUntil(
    caches.keys()
      .then((cacheNames) => {
        return Promise.all(
          cacheNames.map((cacheName) => {
            if (cacheName !== STATIC_CACHE_NAME && cacheName !== DYNAMIC_CACHE_NAME) {
              console.log('Deleting old cache:', cacheName);
              return caches.delete(cacheName);
```

```
          }
        })
      );
    })
    .then(() => {
      console.log('Service Worker activated');
      return self.clients.claim();
    })
  );
});

// Fetch event with advanced caching strategy
self.addEventListener('fetch', (event) => {
  const { request } = event;
  const url = new URL(request.url);

  // Skip non-GET requests
  if (request.method !== 'GET') {
    return;
  }

  // Skip cross-origin requests
  if (url.origin !== location.origin) {
    return;
  }

  event.respondWith(
    caches.match(request)
      .then((cachedResponse) => {
        // If we have a cached response, return it
        if (cachedResponse) {
          return cachedResponse;
        }

        // For API requests, try network first
        if (url.pathname.startsWith('/api/')) {
          return networkFirst(request);
        }

        // For static assets, cache first
        if (isStaticAsset(url.pathname)) {
          return cacheFirst(request);
        }

        // For pages, stale while revalidate
        return staleWhileRevalidate(request);
      })
```

```
      .catch(() => {
        // Return offline page for navigation requests
        if (request.destination === 'document') {
          return caches.match('/offline.html');
        }
      })
  );
});

// Network first strategy for API calls
async function networkFirst(request) {
  try {
    const networkResponse = await fetch(request);

    if (networkResponse.ok) {
      // Cache successful API responses
      const cache = await caches.open(DYNAMIC_CACHE_NAME);
      cache.put(request, networkResponse.clone());
      limitCacheSize(DYNAMIC_CACHE_NAME, DYNAMIC_CACHE_LIMIT);
    }

    return networkResponse;
  } catch (error) {
    console.log('Network failed, trying cache:', error);
    const cachedResponse = await caches.match(request);

    if (cachedResponse) {
      return cachedResponse;
    }

    throw error;
  }
}

// Cache first strategy for static assets
async function cacheFirst(request) {
  const cachedResponse = await caches.match(request);

  if (cachedResponse) {
    return cachedResponse;
  }

  try {
    const networkResponse = await fetch(request);

    if (networkResponse.ok) {
      const cache = await caches.open(STATIC_CACHE_NAME);
```

```
      cache.put(request, networkResponse.clone());
    }

    return networkResponse;
  } catch (error) {
    console.log('Failed to fetch static asset:', error);
    throw error;
  }
}

// Stale while revalidate strategy for pages
async function staleWhileRevalidate(request) {
  const cache = await caches.open(DYNAMIC_CACHE_NAME);
  const cachedResponse = await cache.match(request);

  const fetchPromise = fetch(request).then((networkResponse) => {
    if (networkResponse.ok) {
      cache.put(request, networkResponse.clone());
      limitCacheSize(DYNAMIC_CACHE_NAME, DYNAMIC_CACHE_LIMIT);
    }
    return networkResponse;
  });

  return cachedResponse || fetchPromise;
}

// Helper function to check if a path is a static asset
function isStaticAsset(pathname) {
  return pathname.startsWith('/static/') ||
         pathname.startsWith('/icons/') ||
         pathname.includes('.css') ||
         pathname.includes('.js') ||
         pathname.includes('.png') ||
         pathname.includes('.jpg') ||
         pathname.includes('.jpeg') ||
         pathname.includes('.svg');
}

// Limit cache size
async function limitCacheSize(cacheName, maxSize) {
  const cache = await caches.open(cacheName);
  const keys = await cache.keys();

  if (keys.length > maxSize) {
    const deletePromises = keys
      .slice(maxSize)
      .map(key => cache.delete(key));
```

```
    await Promise.all(deletePromises);
  }
}

// Background sync for offline prompt generation
self.addEventListener('sync', (event) => {
  console.log('Background sync triggered:', event.tag);

  if (event.tag === 'background-sync-prompts') {
    event.waitUntil(syncOfflinePrompts());
  }
});

async function syncOfflinePrompts() {
  try {
    const offlinePrompts = await getOfflinePrompts();
    console.log('Syncing offline prompts:', offlinePrompts.length);

    for (const prompt of offlinePrompts) {
      try {
        const response = await fetch('/api/prompts/generate', {
          method: 'POST',
          headers: {
            'Content-Type': 'application/json',
            'Authorization': `Bearer ${prompt.token}`
          },
          body: JSON.stringify(prompt.data)
        });

        if (response.ok) {
          console.log('Prompt synced successfully:', prompt.id);
          await removeOfflinePrompt(prompt.id);

          // Notify the client
          const clients = await self.clients.matchAll();
          clients.forEach(client => {
            client.postMessage({
              type: 'PROMPT_SYNCED',
              payload: { promptId: prompt.id }
            });
          });
        }
      } catch (error) {
        console.error('Failed to sync prompt:', prompt.id, error);
      }
    }
```

```
  } catch (error) {
    console.error('Background sync failed:', error);
  }
}

// IndexedDB operations for offline prompts
async function getOfflinePrompts() {
  return new Promise((resolve, reject) => {
    const request = indexedDB.open('PromptEngineering', 1);

    request.onerror = () => reject(request.error);

    request.onsuccess = () => {
      const db = request.result;
      const transaction = db.transaction(['offlinePrompts'], 'readonly');
      const store = transaction.objectStore('offlinePrompts');
      const getAllRequest = store.getAll();

      getAllRequest.onsuccess = () => resolve(getAllRequest.result);
      getAllRequest.onerror = () => reject(getAllRequest.error);
    };

    request.onupgradeneeded = () => {
      const db = request.result;
      if (!db.objectStoreNames.contains('offlinePrompts')) {
        db.createObjectStore('offlinePrompts', { keyPath: 'id' });
      }
    };
  });
}

async function removeOfflinePrompt(id) {
  return new Promise((resolve, reject) => {
    const request = indexedDB.open('PromptEngineering', 1);

    request.onsuccess = () => {
      const db = request.result;
      const transaction = db.transaction(['offlinePrompts'], 'readwrite');
      const store = transaction.objectStore('offlinePrompts');
      const deleteRequest = store.delete(id);

      deleteRequest.onsuccess = () => resolve();
      deleteRequest.onerror = () => reject(deleteRequest.error);
    };
  });
}
```

```
// Push notification handling
self.addEventListener('push', (event) => {
  if (!event.data) return;

  const data = event.data.json();

  const options = {
    body: data.body,
    icon: '/icons/icon-192x192.png',
    badge: '/icons/badge-72x72.png',
    tag: data.tag || 'default',
    data: data.data || {},
    actions: data.actions || [],
    vibrate: [200, 100, 200],
    requireInteraction: data.requireInteraction || false
  };

  event.waitUntil(
    self.registration.showNotification(data.title, options)
  );
});

// Notification click handling
self.addEventListener('notificationclick', (event) => {
  event.notification.close();

  let clickAction = event.action;

  if (!clickAction) {
    clickAction = 'default';
  }

  switch (clickAction) {
    case 'view_prompt':
      event.waitUntil(
        clients.openWindow(`/prompts/${event.notification.data.promptId}`)
      );
      break;
    case 'view_community':
      event.waitUntil(
        clients.openWindow('/community')
      );
      break;
    default:
      event.waitUntil(
        clients.openWindow('/')
      );
```

```
  }
});

console.log('Service Worker script loaded');
```

🚀 FAZ 3: Gelişmiş Özellikler (3-4 Hafta)

3.1 AI Destekli Prompt Optimizasyonu

typescript

```typescript
// src/services/promptOptimizer.ts
interface OptimizationStrategy {
  name: string;
  description: string;
  apply: (prompt: string, context?: OptimizationContext) => Promise<string>;
}

interface OptimizationContext {
  targetAudience?: string;
  domain?: string;
  complexity?: 'beginner' | 'intermediate' | 'advanced';
  goals?: string[];
}

interface OptimizationResult {
  original: string;
  optimized: string;
  improvements: {
    clarity: number;
    specificity: number;
    completeness: number;
    effectiveness: number;
    overall: number;
  };
  strategy: string;
  confidence: number;
  suggestions: string[];
  allVersions: OptimizedVersion[];
}

interface OptimizedVersion {
  prompt: string;
  strategy: string;
  performance: PerformanceMetrics;
  improvement: number;
}

interface PerformanceMetrics {
  clarity: number;
  specificity: number;
  completeness: number;
  effectiveness: number;
  overall: number;
  confidence: number;
}
```

```
class PromptOptimizer {
  private strategies: OptimizationStrategy[];
  private mlModel: OptimizationModel;
  private aiProvider: AIProviderManager;

  constructor() {
    this.aiProvider = new AIProviderManager();
    this.strategies = [
      {
        name: 'clarity_enhancement',
        description: 'Netlik ve anlaşılabilirlik iyileştirmesi',
        apply: this.enhanceClarity.bind(this)
      },
      {
        name: 'specificity_increase',
        description: 'Spesifiklik ve detay artırımı',
        apply: this.increaseSpecificity.bind(this)
      },
      {
        name: 'context_enrichment',
        description: 'Bağlam zenginleştirmesi',
        apply: this.enrichContext.bind(this)
      },
      {
        name: 'structure_improvement',
        description: 'Yapı ve format iyileştirmesi',
        apply: this.improveStructure.bind(this)
      },
      {
        name: 'example_addition',
        description: 'Örnek ve rehberlik ekleme',
        apply: this.addExamples.bind(this)
      },
      {
        name: 'constraint_clarification',
        description: 'Kısıtlama ve sınır belirleme',
        apply: this.clarifyConstraints.bind(this)
      }
    ];
  }

  async optimizePrompt(
    originalPrompt: string,
    context?: OptimizationContext
  ): Promise<OptimizationResult> {

    console.log('Starting prompt optimization for:', originalPrompt.substring(0, 100) + '...');
```

```
// Baseline performansı analiz et
const baseline = await this.analyzePromptPerformance(originalPrompt);

const optimizedVersions: OptimizedVersion[] = [];

// Her stratejiyi uygula
for (const strategy of this.strategies) {
 try {
  console.log(`Applying strategy: ${strategy.name}`);

  const optimized = await strategy.apply(originalPrompt, context);
  const performance = await this.analyzePromptPerformance(optimized);

  const improvement = this.calculateImprovement(baseline, performance);

  optimizedVersions.push({
   prompt: optimized,
   strategy: strategy.name,
   performance,
   improvement
  });

  console.log(`Strategy ${strategy.name} improvement: ${improvement.toFixed(2)}`);
 } catch (error) {
  console.error(`Strategy ${strategy.name} failed:`, error);
 }
}

// En iyi performansı seç
const bestVersion = optimizedVersions.reduce((best, current) =>
 current.improvement > best.improvement ? current : best
);

// Genel öneriler oluştur
const suggestions = await this.generateSuggestions(originalPrompt, bestVersion.prompt);

const result: OptimizationResult = {
 original: originalPrompt,
 optimized: bestVersion.prompt,
 improvements: {
  clarity: bestVersion.performance.clarity - baseline.clarity,
  specificity: bestVersion.performance.specificity - baseline.specificity,
  completeness: bestVersion.performance.completeness - baseline.completeness,
  effectiveness: bestVersion.performance.effectiveness - baseline.effectiveness,
  overall: bestVersion.improvement
 },
```

```
    strategy: bestVersion.strategy,
    confidence: bestVersion.performance.confidence,
    suggestions,
    allVersions: optimizedVersions
  };

  console.log('Optimization completed. Best strategy:', bestVersion.strategy);
  return result;
 }

 private async enhanceClarity(prompt: string, context?: OptimizationContext): Promise<string> {
  const optimizationPrompt = `
Sen bir prompt optimizasyon uzmanısın. Aşağıdaki prompt'u daha net ve anlaşılır hale getir:

MEVCUT PROMPT:
"""
${prompt}
"""

İYİLEŞTİRME KRİTERLERİ:
- Belirsizlikleri gider ve net talimatlar ver
- Karmaşık cümleleri sadeleştir
- Gereksiz tekrarları kaldır
- Mantıklı sıralama kullan
- Her adımı açık bir şekilde tanımla

${context ? `
BAĞLAM BİLGİSİ:
- Hedef kitle: ${context.targetAudience || 'Genel'}
- Seviye: ${context.complexity || 'Orta'}
- Alan: ${context.domain || 'Genel'}
` : ''}

İyileştirilmiş prompt'u SADECE metni vererek yanıtla, ek açıklama yapma:`;

  const result = await this.aiProvider.generatePrompt(optimizationPrompt, {
    model: 'claude4',
    technique: 'cot',
    maxTokens: 2000
  });

  return result.content.trim();
 }

 private async increaseSpecificity(prompt: string, context?: OptimizationContext): Promise<string> {
  const optimizationPrompt = `
Bu prompt'u daha spesifik ve hedefli hale getir:
```

```
MEVCUT PROMPT:
"""
${prompt}
"""

SPESİFİKLİK KRİTERLERİ:
- Concrete örnekler ve senaryolar ekle
- Ölçülebilir çıktı kriterleri tanımla
- Kesin format ve yapı belirle
- Beklentileri sayısal olarak ifade et
- Başarı kriterlerini net bir şekilde tanımla

${context?.goals ? `
HEDEFLER: ${context.goals.join(', ')}
` : ''}

Spesifik prompt'u SADECE metni vererek yanıtla:`;

    const result = await this.aiProvider.generatePrompt(optimizationPrompt, {
      model: 'claude4',
      technique: 'cot',
      maxTokens: 2000
    });

    return result.content.trim();
  }

  private async enrichContext(prompt: string, context?: OptimizationContext): Promise<string> {
    const optimizationPrompt = `
Bu prompt'a daha fazla bağlam ve rehberlik ekle:

MEVCUT PROMPT:
"""
${prompt}
"""

BAĞLAM ZENGİNLEŞTİRME:
- Gerekli background bilgisini ekle
- Hedef kitle ve kullanım amacını belirt
- Sınırlamalar ve kısıtları tanımla
- Beklenen kalite standartlarını açıkla
- İlgili örnekler ve referanslar ver

${context ? `
EK BAĞLAM:
- Domain: ${context.domain}
```

```
- Hedef kitle: ${context.targetAudience}
- Karmaşıklık: ${context.complexity}
` : ''}

Bağlam zenginleştirilmiş prompt'u yanıtla:`;

    const result = await this.aiProvider.generatePrompt(optimizationPrompt, {
      model: 'claude4',
      technique: 'constitutional',
      maxTokens: 2500
    });

    return result.content.trim();
  }

  private async improveStructure(prompt: string): Promise<string> {
    const optimizationPrompt = `
Bu prompt'un yapısını ve formatını iyileştir:

MEVCUT PROMPT:
"""
${prompt}
"""

YAPI İYİLEŞTİRMELERİ:
- Başlıklar ve alt başlıklar kullan
- Madde işaretleri ile düzenle
- Adım adım sıralama yap
- Önemli noktaları vurgula
- Kolay okunabilir format kullan

Yapısal olarak iyileştirilmiş prompt'u yanıtla:`;

    const result = await this.aiProvider.generatePrompt(optimizationPrompt, {
      model: 'gpt4o',
      technique: 'meta',
      maxTokens: 2000
    });

    return result.content.trim();
  }

  private async addExamples(prompt: string, context?: OptimizationContext): Promise<string> {
    const optimizationPrompt = `
Bu prompt'a yararlı örnekler ve rehberlik ekle:

MEVCUT PROMPT:
```

```
"""
${prompt}
"""

ÖRNEK EKLEME KRİTERLERİ:
- Pratik kullanım örnekleri ver
- İyi ve kötü örnekleri karşılaştır
- Çıktı format örnekleri göster
- Edge case'leri örneklerle açıkla
- Template'ler ve şablonlar ekle

Örneklerle zenginleştirilmiş prompt'u yanıtla:`;

    const result = await this.aiProvider.generatePrompt(optimizationPrompt, {
      model: 'claude4',
      technique: 'tot',
      maxTokens: 3000
    });

    return result.content.trim();
  }

  private async clarifyConstraints(prompt: string): Promise<string> {
    const optimizationPrompt = `
Bu prompt'a net kısıtlamalar ve sınırlar ekle:

MEVCUT PROMPT:
"""
${prompt}
"""

KISITLAMA BELİRLEME:
- Yapılması ve yapılmaması gerekenleri listele
- Token/kelime sınırlarını belirt
- Kalite kriterlerini tanımla
- Zaman kısıtlarını açıkla
- Güvenlik ve etik sınırları koy

Kısıtlamalarla netleştirilmiş prompt'u yanıtla:`;

    const result = await this.aiProvider.generatePrompt(optimizationPrompt, {
      model: 'claude4',
      technique: 'constitutional',
      maxTokens: 2000
    });

    return result.content.trim();
```

```
}

private async analyzePromptPerformance(prompt: string): Promise<PerformanceMetrics> {
 // Çoklu metrik analizi
 const [clarity, specificity, completeness, effectiveness] = await Promise.all([
  this.calculateClarityScore(prompt),
  this.calculateSpecificityScore(prompt),
  this.calculateCompletenessScore(prompt),
  this.calculateEffectivenessScore(prompt)
 ]);

 const overall = (clarity + specificity + completeness + effectiveness) / 4;
 const confidence = this.calculateConfidence([clarity, specificity, completeness, effectiveness]);

 return {
  clarity,
  specificity,
  completeness,
  effectiveness,
  overall,
  confidence
 };
}

private async calculateClarityScore(prompt: string): Promise<number> {
 // Netlik skorunu hesapla
 const factors = {
  sentenceLength: this.analyzeSentenceLength(prompt),
  vocabularyComplexity: this.analyzeVocabulary(prompt),
  structuralClarity: this.analyzeStructure(prompt),
  ambiguityLevel: this.analyzeAmbiguity(prompt)
 };

 return (factors.sentenceLength + factors.vocabularyComplexity +
      factors.structuralClarity + (1 - factors.ambiguityLevel)) / 4;
}

private async calculateSpecificityScore(prompt: string): Promise<number> {
 // Spesifiklik skorunu hesapla
 const specificityIndicators = [
  /\d+/g.test(prompt), // Sayısal değerler
  /örnek|example/gi.test(prompt), // Örnekler
  /format|şablon|template/gi.test(prompt), // Format belirtimleri
  /kriter|standart|requirement/gi.test(prompt), // Kriterler
  /adım|step|süreç/gi.test(prompt) // Süreç tanımları
 ];
```

```
    return specificityIndicators.filter(Boolean).length / specificityIndicators.length;
  }

  private async calculateCompletenessScore(prompt: string): Promise<number> {
    // Tamlık skorunu hesapla
    const completenessChecks = [
      /ne yapılacak|what to do|görev|task/gi.test(prompt), // Görev tanımı
      /nasıl yapılacak|how to|yöntem|method/gi.test(prompt), // Yöntem
      /neden|why|amaç|purpose/gi.test(prompt), // Amaç
      /çıktı|output|sonuç|result/gi.test(prompt), // Beklenen çıktı
      /sınır|limit|kısıt|constraint/gi.test(prompt) // Sınırlamalar
    ];

    return completenessChecks.filter(Boolean).length / completenessChecks.length;
  }

  private async calculateEffectivenessScore(prompt: string): Promise<number> {
    // Etkililik skorunu AI ile değerlendir
    const evaluationPrompt = `
Aşağıdaki prompt'un etkililik seviyesini 0-1 arasında değerlendir:

"""
${prompt}
"""

Değerlendirme kriterleri:
- AI'ın doğru ve faydalı yanıt verebilme olasılığı
- Prompt'un açıklığı ve anlaşılabilirliği
- Beklenen sonuçları alma potansiyeli
- Belirsizlik ve yanıltıcılık seviyesi

SADECE 0.0 ile 1.0 arasında bir sayı yanıtla:`;

    try {
      const result = await this.aiProvider.generatePrompt(evaluationPrompt, {
        model: 'claude4',
        technique: 'cot',
        maxTokens: 100
      });

      const score = parseFloat(result.content.trim());
      return isNaN(score) ? 0.5 : Math.max(0, Math.min(1, score));
    } catch (error) {
      console.error('Effectiveness calculation failed:', error);
      return 0.5; // Default score
    }
  }
```

```
private analyzeSentenceLength(prompt: string): number {
  const sentences = prompt.split(/[.!?]+/).filter(s => s.trim().length > 0);
  const avgLength = sentences.reduce((sum, s) => sum + s.length, 0) / sentences.length;

  // Optimal sentence length: 15-25 words (roughly 75-125 characters)
  const optimalMin = 75;
  const optimalMax = 125;

  if (avgLength >= optimalMin && avgLength <= optimalMax) {
    return 1.0;
  } else if (avgLength < optimalMin) {
    return Math.max(0.3, avgLength / optimalMin);
  } else {
    return Math.max(0.3, optimalMax / avgLength);
  }
}

private analyzeVocabulary(prompt: string): number {
  const words = prompt.toLowerCase().match(/\b\w+\b/g) || [];
  const complexWords = words.filter(word => word.length > 8).length;
  const complexity = complexWords / words.length;

  // Optimal complexity: 10-25% complex words
  if (complexity >= 0.1 && complexity <= 0.25) {
    return 1.0;
  } else if (complexity < 0.1) {
    return Math.max(0.5, complexity / 0.1);
  } else {
    return Math.max(0.3, 0.25 / complexity);
  }
}

private analyzeStructure(prompt: string): number {
  const structureIndicators = [
    /^\d+\./gm.test(prompt), // Numbered lists
    /^[-*•]/gm.test(prompt), // Bullet points
    /^#{1,6}\s/gm.test(prompt), // Headers
    /\n\n/g.test(prompt), // Paragraphs
    /[:\-]{2,}/g.test(prompt) // Separators
  ];

  return structureIndicators.filter(Boolean).length / structureIndicators.length;
}

private analyzeAmbiguity(prompt: string): number {
  const ambiguousTerms = [
```

```
    /belki|maybe|perhaps|possibly/gi,
    /biraz|somewhat|kind of|sort of/gi,
    /yaklaşık|approximately|around|about/gi,
    /genellikle|usually|generally|typically/gi,
    /bazı|some|several|various/gi
  ];

  const ambiguityCount = ambiguousTerms.reduce((count, regex) => {
    return count + (prompt.match(regex) || []).length;
  }, 0);

  const words = prompt.split(/\s+/).length;
  return Math.min(1, ambiguityCount / words * 10); // Normalize to 0-1
}

private calculateConfidence(scores: number[]): number {
  const variance = this.calculateVariance(scores);
  const avgScore = scores.reduce((sum, score) => sum + score, 0) / scores.length;

  // Lower variance = higher confidence
  const normalizedVariance = Math.min(1, variance * 4);
  const confidence = (1 - normalizedVariance) * avgScore;

  return Math.max(0.1, Math.min(1, confidence));
}

private calculateVariance(numbers: number[]): number {
  const mean = numbers.reduce((sum, num) => sum + num, 0) / numbers.length;
  const squaredDiffs = numbers.map(num => Math.pow(num - mean, 2));
  return squaredDiffs.reduce((sum, diff) => sum + diff, 0) / numbers.length;
}

private calculateImprovement(baseline: PerformanceMetrics, optimized: PerformanceMetrics): number {
  return optimized.overall - baseline.overall;
}

private async generateSuggestions(original: string, optimized: string): Promise<string[]> {
  const suggestionPrompt = `
Orijinal ve optimize edilmiş prompt'ları karşılaştırarak iyileştirme önerileri oluştur:

ORİJİNAL:
"""
${original}
"""

OPTİMİZE EDİLMİŞ:
"""
```

```
${optimized}
"""

5 adet kısa ve pratik iyileştirme önerisi ver. Her öneri tek satırda olsun ve spesifik olsun:`;

    try {
      const result = await this.aiProvider.generatePrompt(suggestionPrompt, {
        model: 'claude4',
        technique: 'meta',
        maxTokens: 500
      });

      return result.content
        .split('\n')
        .filter(line => line.trim().length > 0)
        .map(line => line.replace(/^\d+\.\s*/, '').trim())
        .slice(0, 5);
    } catch (error) {
      console.error('Suggestion generation failed:', error);
      return [
        'Daha spesifik talimatlar ekleyin',
        'Örnek çıktılar gösterin',
        'Sınırlamaları net bir şekilde belirtin',
        'Adım adım süreç tanımlayın',
        'Kalite kriterlerini açıklayın'
      ];
    }
  }
}

// A/B Testing sistemi
class PromptABTesting {
  private experiments: Map<string, Experiment> = new Map();

  async createExperiment(
    basePrompt: string,
    variations: string[],
    testConfig: TestConfig
  ): Promise<Experiment> {

    const experiment: Experiment = {
      id: this.generateId(),
      name: testConfig.name,
      basePrompt,
      variations: [
        { id: 'control', prompt: basePrompt, traffic: 0.5 },
        ...variations.map((v, i) => ({
```

```
        id: `variant_${i + 1}`,
        prompt: v,
        traffic: 0.5 / variations.length
      }))
    ],
    config: testConfig,
    status: 'running',
    createdAt: new Date(),
    results: [],
    metrics: {
      conversions: new Map(),
      impressions: new Map(),
      ratings: new Map()
    }
  };

  this.experiments.set(experiment.id, experiment);

  console.log('A/B Test created:', experiment.id);
  return experiment;
}

async getVariationForUser(experimentId: string, userId: string): Promise<string> {
  const experiment = this.experiments.get(experimentId);
  if (!experiment || experiment.status !== 'running') {
    throw new Error('Experiment not found or not running');
  }

  // Consistent hashing for user assignment
  const hash = this.hashUserId(userId + experimentId);
  let cumulative = 0;

  for (const variation of experiment.variations) {
    cumulative += variation.traffic;
    if (hash < cumulative) {

      // Record impression
      const currentImpressions = experiment.metrics.impressions.get(variation.id) || 0;
      experiment.metrics.impressions.set(variation.id, currentImpressions + 1);

      return variation.prompt;
    }
  }

  return experiment.basePrompt; // Fallback
}
```

```
async recordResult(
  experimentId: string,
  variationId: string,
  metrics: ResultMetrics
): Promise<void> {

  const experiment = this.experiments.get(experimentId);
  if (!experiment) {
    throw new Error('Experiment not found');
  }

  const result: ExperimentResult = {
    experimentId,
    variationId,
    metrics,
    timestamp: new Date()
  };

  experiment.results.push(result);

  // Update aggregated metrics
  if (metrics.converted) {
    const currentConversions = experiment.metrics.conversions.get(variationId) || 0;
    experiment.metrics.conversions.set(variationId, currentConversions + 1);
  }

  if (metrics.rating) {
    const currentRatings = experiment.metrics.ratings.get(variationId) || [];
    currentRatings.push(metrics.rating);
    experiment.metrics.ratings.set(variationId, currentRatings);
  }

  // Check for statistical significance
  const analysis = await this.analyzeResults(experiment);

  if (analysis.isSignificant && experiment.config.autoStop) {
    experiment.status = 'completed';
    console.log('Experiment auto-stopped due to significance:', experimentId);

    await this.notifySignificantResult(experiment, analysis);
  }
}

private async analyzeResults(experiment: Experiment): Promise<StatisticalAnalysis> {
  const variations = experiment.variations;
  const results = experiment.results;
```

```
  if (results.length < experiment.config.minSampleSize) {
    return {
      isSignificant: false,
      confidence: 0,
      winningVariation: null,
      effectSize: 0,
      recommendedAction: 'continue'
    };
  }

  // Calculate conversion rates
  const conversionRates = new Map<string, number>();
  const sampleSizes = new Map<string, number>();

  for (const variation of variations) {
    const variationResults = results.filter(r => r.variationId === variation.id);
    const conversions = variationResults.filter(r => r.metrics.converted).length;
    const samples = variationResults.length;

    conversionRates.set(variation.id, samples > 0 ? conversions / samples : 0);
    sampleSizes.set(variation.id, samples);
  }

  // Perform Chi-square test
  const chiSquareResult = this.performChiSquareTest(conversionRates, sampleSizes);

  // Find best performing variation
  let bestVariation = variations[0].id;
  let bestRate = conversionRates.get(bestVariation) || 0;

  for (const [varId, rate] of conversionRates) {
    if (rate > bestRate) {
      bestRate = rate;
      bestVariation = varId;
    }
  }

  return {
    isSignificant: chiSquareResult.pValue < 0.05,
    confidence: 1 - chiSquareResult.pValue,
    winningVariation: bestVariation,
    effectSize: this.calculateEffectSize(conversionRates),
    recommendedAction: this.getRecommendation(chiSquareResult, experiment.config)
  };
}

private performChiSquareTest(
```

```
    conversionRates: Map<string, number>,
    sampleSizes: Map<string, number>
  ): ChiSquareResult {
    // Simplified Chi-square test implementation
    let chiSquare = 0;
    let totalObserved = 0;
    let totalExpected = 0;

    const variations = Array.from(conversionRates.keys());

    for (const varId of variations) {
      const rate = conversionRates.get(varId) || 0;
      const samples = sampleSizes.get(varId) || 0;
      const observed = rate * samples;
      const expected = samples * 0.5; // Assume 50% baseline

      if (expected > 0) {
        chiSquare += Math.pow(observed - expected, 2) / expected;
      }

      totalObserved += observed;
      totalExpected += expected;
    }

    // Degrees of freedom
    const df = variations.length - 1;

    // Convert chi-square to p-value (simplified)
    const pValue = this.chiSquareToPValue(chiSquare, df);

    return {
      chiSquare,
      pValue,
      degreesOfFreedom: df,
      bestVariation: this.findBestVariation(conversionRates)
    };
  }

  private chiSquareToPValue(chiSquare: number, df: number): number {
    // Simplified p-value calculation
    // In production, use a proper statistical library
    if (df === 1) {
      if (chiSquare > 3.84) return 0.05;
      if (chiSquare > 6.64) return 0.01;
      if (chiSquare > 10.83) return 0.001;
    }
```

```
  return chiSquare > 3.84 ? 0.05 : 0.1; // Simplified
}

private findBestVariation(conversionRates: Map<string, number>): string {
  let best = '';
  let bestRate = 0;

  for (const [varId, rate] of conversionRates) {
    if (rate > bestRate) {
      bestRate = rate;
      best = varId;
    }
  }

  return best;
}

private calculateEffectSize(conversionRates: Map<string, number>): number {
  const rates = Array.from(conversionRates.values());
  const max = Math.max(...rates);
  const min = Math.min(...rates);

  return min > 0 ? (max - min) / min : 0;
}

private getRecommendation(
  result: ChiSquareResult,
  config: TestConfig
): 'continue' | 'stop' | 'extend' {
  if (result.pValue < 0.05) {
    return 'stop';
  } else if (result.pValue < 0.1) {
    return 'extend';
  } else {
    return 'continue';
  }
}

private hashUserId(input: string): number {
  let hash = 0;
  for (let i = 0; i < input.length; i++) {
    const char = input.charCodeAt(i);
    hash = ((hash << 5) - hash) + char;
    hash = hash & hash; // Convert to 32-bit integer
  }
  return Math.abs(hash) % 1000 / 1000; // Normalize to 0-1
}
```

```
  private generateId(): string {
    return 'exp_' + Date.now().toString(36) + Math.random().toString(36).substr(2);
  }

  private async notifySignificantResult(
    experiment: Experiment,
    analysis: StatisticalAnalysis
  ): Promise<void> {
    console.log('Significant A/B test result:', {
      experimentId: experiment.id,
      winningVariation: analysis.winningVariation,
      confidence: analysis.confidence
    });

    // In production, send notifications via email, Slack, etc.
  }
}

// Types
interface TestConfig {
  name: string;
  minSampleSize: number;
  maxDuration: number; // days
  significanceLevel: number;
  autoStop: boolean;
}

interface Experiment {
  id: string;
  name: string;
  basePrompt: string;
  variations: PromptVariation[];
  config: TestConfig;
  status: 'running' | 'paused' | 'completed';
  createdAt: Date;
  results: ExperimentResult[];
  metrics: {
    conversions: Map<string, number>;
    impressions: Map<string, number>;
    ratings: Map<string, number[]>;
  };
}

interface PromptVariation {
  id: string;
  prompt: string;
```

```
  traffic: number; // 0-1
}

interface ExperimentResult {
  experimentId: string;
  variationId: string;
  metrics: ResultMetrics;
  timestamp: Date;
}

interface ResultMetrics {
  converted: boolean;
  rating?: number;
  latency?: number;
  tokenUsage?: number;
  userSatisfaction?: number;
}

interface StatisticalAnalysis {
  isSignificant: boolean;
  confidence: number;
  winningVariation: string | null;
  effectSize: number;
  recommendedAction: 'continue' | 'stop' | 'extend';
}

interface ChiSquareResult {
  chiSquare: number;
  pValue: number;
  degreesOfFreedom: number;
  bestVariation: string;
}

export { PromptOptimizer, PromptABTesting };
```

3.2 Topluluk ve Sosyal Özellikler

typescript

```tsx
// src/components/Community/CommunityHub.tsx
import React, { useState, useCallback } from 'react';
import { motion, AnimatePresence } from 'framer-motion';
import { useQuery, useInfiniteQuery } from '@tanstack/react-query';
import { useVirtualizer } from '@tanstack/react-virtual';
import { communityService } from '../../services/communityService';
import { PromptCard } from './PromptCard';
import { SearchFilters } from './SearchFilters';
import { SortOptions } from './SortOptions';
import { TrendingPrompts } from './TrendingPrompts';
import { LoadingSpinner } from '../ui/LoadingSpinner';
import { useDebounce } from '../../hooks/useDebounce';
import { useIntersectionObserver } from '../../hooks/useIntersectionObserver';

interface CommunityFilters {
  technique: string;
  model: string;
  tag: string;
  rating: number;
  dateRange: string;
  search: string;
}

export const CommunityHub: React.FC = () => {
  const [filters, setFilters] = useState<CommunityFilters>({
    technique: '',
    model: '',
    tag: '',
    rating: 0,
    dateRange: 'all',
    search: ''
  });

  const [sortBy, setSortBy] = useState('trending');
  const [viewMode, setViewMode] = useState<'grid' | 'list'>('list');

  const debouncedSearch = useDebounce(filters.search, 300);
  const { targetRef, isIntersecting } = useIntersectionObserver();

  // Infinite query for prompts
  const {
    data: promptsData,
    fetchNextPage,
    hasNextPage,
    isFetchingNextPage,
    isLoading,
```

```
  error
} = useInfiniteQuery({
  queryKey: ['community-prompts', filters, sortBy, debouncedSearch],
  queryFn: ({ pageParam = 1 }) =>
    communityService.getPublicPrompts({
      ...filters,
      search: debouncedSearch,
      sortBy,
      page: pageParam
    }),
  getNextPageParam: (lastPage) =>
    lastPage.pagination.page < lastPage.pagination.pages
      ? lastPage.pagination.page + 1
      : undefined,
  staleTime: 5 * 60 * 1000 // 5 minutes
});

// Trending prompts
const { data: trending } = useQuery({
  queryKey: ['trending-prompts'],
  queryFn: () => communityService.getTrendingPrompts(),
  refetchInterval: 5 * 60 * 1000, // 5 dakikada bir güncelle
  staleTime: 2 * 60 * 1000 // 2 minutes
});

// Featured prompts
const { data: featured } = useQuery({
  queryKey: ['featured-prompts'],
  queryFn: () => communityService.getFeaturedPrompts(),
  staleTime: 10 * 60 * 1000 // 10 minutes
});

// Load more when scrolling to bottom
React.useEffect(() => {
  if (isIntersecting && hasNextPage && !isFetchingNextPage) {
    fetchNextPage();
  }
}, [isIntersecting, hasNextPage, isFetchingNextPage, fetchNextPage]);

const handleFilterChange = useCallback((newFilters: Partial<CommunityFilters>) => {
  setFilters(prev => ({ ...prev, ...newFilters }));
}, []);

const clearFilters = useCallback(() => {
  setFilters({
    technique: '',
    model: '',
```

```
    tag: '',
    rating: 0,
    dateRange: 'all',
    search: ''
  });
}, []);

// Flatten all pages of prompts
const allPrompts = promptsData?.pages.flatMap(page => page.prompts) || [];
const totalCount = promptsData?.pages[0]?.pagination.total || 0;

if (error) {
  return (
    <div className="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8 py-8">
      <div className="text-center">
        <div className="text-red-500 text-lg mb-4">
          Topluluk prompt'ları yüklenirken hata oluştu
        </div>
        <button
          onClick={() => window.location.reload()}
          className="px-4 py-2 bg-blue-600 text-white rounded-lg hover:bg-blue-700"
        >
          Tekrar Dene
        </button>
      </div>
    </div>
  );
}

return (
  <div className="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8 py-8">
    {/* Header */}
    <motion.div
      className="mb-8"
      initial={{ opacity: 0, y: -20 }}
      animate={{ opacity: 1, y: 0 }}
      transition={{ duration: 0.3 }}
    >
      <h1 className="text-4xl font-bold text-gray-900 mb-2">
        🌟 Topluluk Prompt'ları
      </h1>
      <p className="text-xl text-gray-600">
        Diğer kullanıcılar tarafından oluşturulmuş ve paylaşılmış en iyi prompt'ları keşfedin
      </p>
    </motion.div>

    {/* Featured Prompts */}
```

```
{featured && featured.length > 0 && (
  <motion.section
    className="mb-12"
    initial={{
```