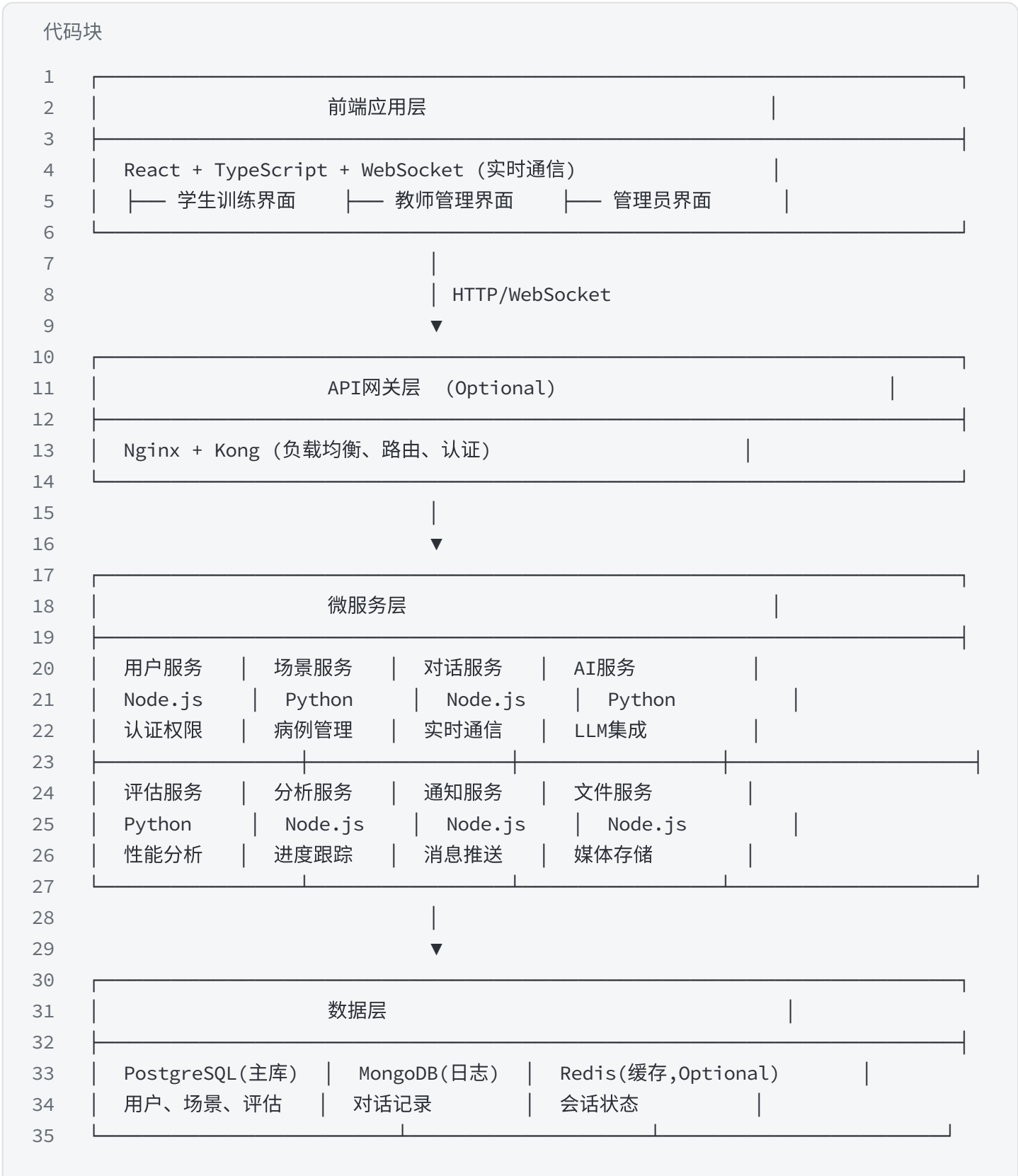
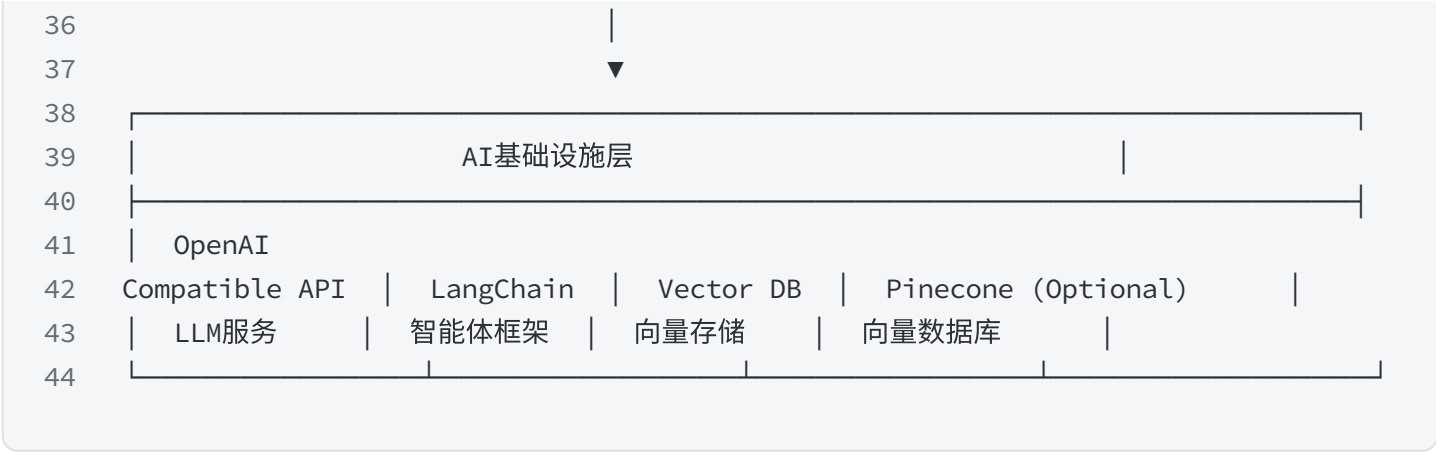


AISP技术架构设计

1.0 整体架构设计

1.1 微服务架构图





1.2 技术栈选择理由

前端技术栈：

代码块

```
1 React + TypeScript + Tailwind CSS + WebSocket
```

选择理由：

- **React**：组件化开发，生态丰富，团队熟悉度高
- **TypeScript**：类型安全，降低运行时错误，提升代码质量
- **Tailwind CSS**：快速开发，设计一致性，减少样式代码
- **WebSocket**：实时通信要求，对话体验流畅

后端技术栈：

代码块

```
1 Node.js（用户/对话/分析）+ Python（AI/评估）+ Express/FastAPI
```

选择理由：

- **Node.js**：高并发处理，WebSocket支持，JavaScript统一技术栈
- **Python**：AI/ML生态丰富，LangChain支持，科学计算库完善
- **Express/FastAPI**：轻量级框架，开发效率高，性能优秀

数据库技术栈：

代码块

```
1 PostgreSQL（主库）+ MongoDB（日志）+ Redis（缓存）
```

选择理由：

- **PostgreSQL**：ACID特性，复杂查询，JSON支持
- **MongoDB**：文档存储，适合对话日志，水平扩展
- **Redis**：内存缓存，会话状态，消息队列

1.3 系统边界定义

核心边界：

- **用户管理**：认证、授权、权限控制
- **场景管理**：病例创建、版本控制、内容审核
- **对话服务**：实时通信、消息路由、状态管理
- **AI服务**：LLM集成、智能体协调、知识管理
- **评估服务**：实时评分、报告生成、趋势分析

外部依赖：

- OpenAI Compatible API (GLM-4)
- SMTP服务 (邮件通知)
- CDN服务 (静态资源)
- 监控服务 (性能跟踪)

2.0 前端架构设计

2.1 React组件架构

代码块

```
1  src/
2  |— components/           # 通用组件
3  |  |— ui/                # 基础UI组件
4  |  |— forms/            # 表单组件
5  |  |— charts/           # 图表组件
6  |— pages/               # 页面组件
7  |  |— student/          # 学生页面
8  |  |— teacher/          # 教师页面
9  |  |— admin/            # 管理员页面
10 |— hooks/               # 自定义Hooks
11 |— services/            # API服务
12 |— store/              # 状态管理
13 |— utils/              # 工具函数
14 |— types/              # TypeScript类型定义
```

2.2 状态管理方案

状态管理架构：

代码块

```
1 Zustand（全局状态） + React Query（服务端状态） + WebSocket（实时状态）
```

状态层次：

1. 全局状态 (Zustand)：

- 用户认证信息
- 应用配置信息
- 主题设置

2. 服务端状态 (React Query)：

- 场景数据
- 用户列表
- 评估报告

3. 实时状态 (WebSocket)：

- 对话消息
- 实时评估
- 在线状态

2.3 WebSocket实时通信设计

连接管理：

代码块

```
1 // WebSocket连接封装
2 class WebSocketService {
3     private ws: WebSocket | null = null;
4     private reconnectAttempts = 0;
5     private maxReconnectAttempts = 5;
6
7     connect(url: string) {
8         this.ws = new WebSocket(url);
9         this.setupEventHandlers();
10    }
11
12    setupEventHandlers() {
```

```

13     this.ws?.addEventListener('open', this.onOpen);
14     this.ws?.addEventListener('message', this.onMessage);
15     this.ws?.addEventListener('error', this.onError);
16     this.ws?.addEventListener('close', this.onClose);
17 }
18
19 sendMessage(type: string, data: any) {
20     this.ws?.send(JSON.stringify({ type, data }));
21 }
22 }

```

消息格式定义:

代码块

```

1 interface WebSocketMessage {
2     type: 'dialogue' | 'assessment' | 'system' | 'heartbeat';
3     sessionId: string;
4     timestamp: number;
5     data: any;
6 }

```

3.0 后端架构设计

3.1 Node.js服务设计

用户服务 (user-service):

代码块

```

1 // 用户服务架构
2 app.use(cors());
3 app.use(helmet());
4 app.use(express.json());
5
6 // 认证中间件
7 app.use(authMiddleware);
8
9 // 路由定义
10 app.use('/api/auth', authRoutes);
11 app.use('/api/users', userRoutes);
12 app.use('/api/roles', roleRoutes);

```

对话服务 (dialogue-service):

代码块 // 对话服务架构

```
1  const io = new Server(server, {
2    cors: { origin: process.env.FRONTEND_URL }
3  });
4
5
6  io.on('connection', (socket) => {
7    socket.on('join_session', handleJoinSession);
8    socket.on('send_message', handleMessage);
9    socket.on('disconnect', handleDisconnect);
10  });
```

3.2 API规范定义

RESTful API设计:

代码块

```
1  # 用户管理
2  GET    /api/users          # 获取用户列表
3  POST   /api/users          # 创建用户
4  GET    /api/users/:id      # 获取用户详情
5  PUT    /api/users/:id      # 更新用户
6  DELETE /api/users/:id      # 删除用户
7
8  # 场景管理
9  GET    /api/scenarios      # 获取场景列表
10 POST   /api/scenarios      # 创建场景
11 GET    /api/scenarios/:id  # 获取场景详情
12 PUT    /api/scenarios/:id  # 更新场景
13
14 # 训练会话
15 POST   /api/sessions        # 创建会话
16 GET    /api/sessions/:id    # 获取会话详情
17 POST   /api/sessions/:id/messages # 发送消息
```

WebSocket事件定义:

代码块

```
1  interface DialogueEvents {
2    'session:join': (sessionId: string) => void;
3    'message:send': (message: DialogueMessage) => void;
4    'assessment:update': (assessment: AssessmentData) => void;
5    'session:end': (result: SessionResult) => void;
6  }
```

3.3 认证授权机制

JWT Token设计:

代码块

```
1 interface JWPayload {
2     userId: string;
3     role: 'student' | 'teacher' | 'admin';
4     permissions: string[];
5     exp: number;
6     iat: number;
7 }
```

权限控制中间件:

代码块

```
1 const authorize = (requiredPermissions: string[]) => {
2     return (req: Request, res: Response, next: NextFunction) => {
3         const token = req.headers.authorization?.split(' ')[1];
4         const decoded = jwt.verify(token, process.env.JWT_SECRET) as JWPayload;
5
6         const hasPermission = requiredPermissions.every(
7             permission => decoded.permissions.includes(permission)
8         );
9
10        if (!hasPermission) {
11            return res.status(403).json({ error: 'Insufficient permissions' });
12        }
13
14        next();
15    };
16 }
```

4.0 AI服务架构

4.1 LLM集成方案

LLM服务封装:

代码块

```
1 import openai
2 from typing import Dict, List, Optional
3
```

```

4  class AIService:
5      def __init__(self):
6          self.client = openai.OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
7          self.model = "gpt-4-turbo-preview"
8
9      async def generate_response(
10         self,
11         messages: List[Dict],
12         temperature: float = 0.7,
13         max_tokens: int = 500
14     ) -> str:
15         response = await self.client.chat.completions.create(
16             model=self.model,
17             messages=messages,
18             temperature=temperature,
19             max_tokens=max_tokens
20         )
21         return response.choices[0].message.content

```

多智能体协调器：

代码块

```

1  from langchain.agents import AgentExecutor, create_openai_tools_agent
2  from langchain_core.prompts import ChatPromptTemplate
3
4  class AgentOrchestrator:
5      def __init__(self):
6          self.patient_agent = self._create_patient_agent()
7          self.knowledge_agent = self._create_knowledge_agent()
8          self.assessment_agent = self._create_assessment_agent()
9          self.feedback_agent = self._create_feedback_agent()
10
11      async def orchestrate_response(
12         self,
13         user_input: str,
14         session_context: Dict
15     ) -> Dict:
16         # 并行调用多个智能体
17         tasks = [
18             self.patient_agent.generate(user_input, session_context),
19             self.knowledge_agent.validate(user_input, session_context),
20             self.assessment_agent.evaluate(user_input, session_context)
21         ]
22
23         results = await asyncio.gather(*tasks)

```

```

24
25         # 合成最终响应
26         final_response = await self.feedback_agent.enhance(
27             results, session_context
28         )
29
30         return final_response

```

4.2 LangChain多智能体设计

病人智能体：

代码块

```

1  PATIENT_PROMPT = ChatPromptTemplate.from_messages([
2      ("system", """
3      你是Maria Rodriguez, 52岁的墨西哥裔教师, 新诊断为2型糖尿病。
4
5      性格特点:
6      - 对新诊断感到焦虑和担忧
7      - 担心像母亲一样出现并发症
8      - 重视传统墨西哥饮食
9      - 家庭意识强, 需要家人支持
10
11     回答要求:
12     - 严格按照JSON剧本内容回应
13     - 体现焦虑但愿意学习的情感
14     - 对文化敏感话题有顾虑
15     - 语言自然, 符合人物背景
16     """),
17      ("human", "{input}")
18  ])

```

医学知识智能体：

代码块

```

1  KNOWLEDGE_PROMPT = ChatPromptTemplate.from_messages([
2      ("system", """
3      你是医学知识验证专家, 专门负责验证糖尿病相关信息的准确性。
4
5      验证标准:
6      - 基于ADA 2023糖尿病管理指南
7      - 确保医学信息准确性 > 98%
8      - 避免提供具体药物剂量建议
9      - 识别潜在的医疗风险

```

```

10
11     输出格式:
12     {{
13         "is_accurate": boolean,
14         "corrections": string[],
15         "risk_level": "low" | "medium" | "high",
16         "suggestions": string[]
17     }}
18     """),
19     ("human", "{input}")
20 ]))

```

4.3 Vector DB设计

向量数据库架构:

代码块

```

1  import pinecone
2  from sentence_transformers import SentenceTransformer
3
4  class VectorDatabase:
5      def __init__(self):
6          self.embedder = SentenceTransformer('all-MiniLM-L6-v2')
7          pinecone.init(api_key=os.getenv("PINECONE_API_KEY"))
8          self.index = pinecone.Index("medical-knowledge")
9
10     def store_knowledge(self, knowledge_id: str, content: str):
11         embedding = self.embedder.encode(content)
12         self.index.upsert(
13             vectors=[{
14                 "id": knowledge_id,
15                 "values": embedding.tolist(),
16                 "metadata": {"content": content}
17             }]
18         )
19
20     def search_knowledge(self, query: str, top_k: int = 5):
21         query_embedding = self.embedder.encode(query)
22         results = self.index.query(
23             vector=query_embedding.tolist(),
24             top_k=top_k
25         )
26         return results

```

5.0 数据架构设计

5.1 PostgreSQL主库设计

数据库连接配置：

代码块

```
1  from sqlalchemy import create_engine
2  from sqlalchemy.ext.declarative import declarative_base
3  from sqlalchemy.orm import sessionmaker
4
5  DATABASE_URL = "postgresql://user:password@localhost:5432/aisp"
6
7  engine = create_engine(DATABASE_URL)
8  SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
9  Base = declarative_base()
```

数据表设计示例：

代码块

```
1  class User(Base):
2      __tablename__ = "users"
3
4      id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
5      username = Column(String(50), unique=True, nullable=False)
6      email = Column(String(100), unique=True, nullable=False)
7      password_hash = Column(String(255), nullable=False)
8      role = Column(Enum('student', 'teacher', 'admin'), nullable=False)
9      created_at = Column(DateTime, default=datetime.utcnow)
10     updated_at = Column(DateTime, default=datetime.utcnow,
        onupdate=datetime.utcnow)
```

5.2 MongoDB日志存储

连接配置：

代码块

```
1  from pymongo import MongoClient
2
3  mongo_client = MongoClient("mongodb://localhost:27017/")
4  db = mongo_client["aisp_logs"]
5  conversations_collection = db["conversations"]
```

对话记录存储：

代码块

```
1 class ConversationLog:
2     def __init__(self):
3         self.collection = conversations_collection
4
5     def store_message(self, session_id: str, message_data: Dict):
6         document = {
7             "session_id": session_id,
8             "timestamp": datetime.utcnow(),
9             "message": message_data
10        }
11        self.collection.insert_one(document)
12
13    def get_conversation_history(self, session_id: str, limit: int = 50):
14        return self.collection.find(
15            {"session_id": session_id}
16        ).sort("timestamp", -1).limit(limit)
```

5.3 Redis缓存策略

缓存配置：

代码块

```
1 import redis
2
3 redis_client = redis.Redis(
4     host='localhost',
5     port=6379,
6     db=0,
7     decode_responses=True
8 )
```

缓存策略设计：

代码块

```
1 class CacheService:
2     def __init__(self):
3         self.redis = redis_client
4         self.default_ttl = 3600 # 1小时
5
```

```

6     def cache_session_state(self, session_id: str, state: Dict):
7         key = f"session:{session_id}:state"
8         self.redis.setex(key, self.default_ttl, json.dumps(state))
9
10    def get_session_state(self, session_id: str) -> Optional[Dict]:
11        key = f"session:{session_id}:state"
12        cached_data = self.redis.get(key)
13        return json.loads(cached_data) if cached_data else None
14
15    def cache_ai_response(self, prompt_hash: str, response: str):
16        key = f"ai:response:{prompt_hash}"
17        self.redis.setex(key, 1800, response) # 30分钟缓存
18
19    def invalidate_user_cache(self, user_id: str):
20        pattern = f"user:{user_id}:*"
21        keys = self.redis.keys(pattern)
22        if keys:
23            self.redis.delete(*keys)

```

6.0 部署架构设计(Optional)

6.1 Docker容器化

多阶段构建Dockerfile:

代码块

```

1  # 前端构建阶段
2  FROM node:18-alpine AS frontend-build
3  WORKDIR /app/frontend
4  COPY frontend/package*.json ./
5  RUN npm ci --only=production
6  COPY frontend/ ./
7  RUN npm run build
8
9  # 后端构建阶段
10 FROM node:18-alpine AS backend-build
11 WORKDIR /app/backend
12 COPY backend/package*.json ./
13 RUN npm ci --only=production
14 COPY backend/ ./
15
16 # 生产镜像
17 FROM node:18-alpine AS production
18 WORKDIR /app
19 COPY --from=frontend-build /app/frontend/dist ./public

```

```
20 COPY --from=backend-build /app/backend .
21 EXPOSE 3000
22 CMD ["npm", "start"]
```

Docker Compose配置：

代码块

```
1  version: '3.8'
2  services:
3    frontend:
4      build: .
5      ports:
6        - "3000:3000"
7      environment:
8        - NODE_ENV=production
9        - API_URL=http://backend:5000
10
11    backend:
12      build: .
13      ports:
14        - "5000:5000"
15      environment:
16        - DATABASE_URL=postgresql://user:pass@postgres:5432/aisp
17        - REDIS_URL=redis://redis:6379
18      depends_on:
19        - postgres
20        - redis
21
22    postgres:
23      image: postgres:15
24      environment:
25        POSTGRES_DB: aisp
26        POSTGRES_USER: user
27        POSTGRES_PASSWORD: pass
28      volumes:
29        - postgres_data:/var/lib/postgresql/data
30
31    redis:
32      image: redis:7-alpine
33      volumes:
34        - redis_data:/data
35
36  volumes:
37    postgres_data:
38    redis_data:
```

6.2 Kubernetes编排

部署配置：

代码块

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: aisp-backend
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: aisp-backend
10   template:
11     metadata:
12       labels:
13         app: aisp-backend
14     spec:
15       containers:
16       - name: backend
17         image: aisp/backend:latest
18         ports:
19         - containerPort: 5000
20         env:
21         - name: DATABASE_URL
22           valueFrom:
23             secretKeyRef:
24               name: aisp-secrets
25               key: database-url
26       resources:
27         requests:
28           memory: "256Mi"
29           cpu: "250m"
30         limits:
31           memory: "512Mi"
32           cpu: "500m"
33   ---
34   apiVersion: v1
35   kind: Service
36   metadata:
37     name: aisp-backend-service
38   spec:
39     selector:
40       app: aisp-backend
41     ports:
```

```
42   - port: 80
43     targetPort: 5000
44     type: ClusterIP
```

6.3 CI/CD流程

GitHub Actions配置：

代码块

```
1  name: CI/CD Pipeline
2
3  on:
4    push:
5      branches: [main]
6    pull_request:
7      branches: [main]
8
9  jobs:
10   test:
11     runs-on: ubuntu-latest
12     steps:
13       - uses: actions/checkout@v3
14       - uses: actions/setup-node@v3
15         with:
16           node-version: '18'
17
18       - name: Install dependencies
19         run: npm ci
20
21       - name: Run tests
22         run: npm test
23
24       - name: Run linting
25         run: npm run lint
26
27       - name: Type checking
28         run: npm run typecheck
29
30   build:
31     needs: test
32     runs-on: ubuntu-latest
33     if: github.ref == 'refs/heads/main'
34
35     steps:
36       - uses: actions/checkout@v3
37
```

```
38     - name: Build Docker image
39       run: docker build -t aisp/backend:${{ github.sha }} .
40
41     - name: Push to registry
42       run: |
43         echo ${ secrets.DOCKER_PASSWORD } | docker login -u ${ secrets.DOCKER_USERNAME } --password-stdin
44         docker push aisp/backend:${{ github.sha }}
45
46     - name: Deploy to Kubernetes
47       run: |
48         kubectl set image deployment/aisp-backend backend=aisp/backend:${{ github.sha }}
49         kubectl rollout status deployment/aisp-backend
```

7.0 监控与安全(Optional)

7.1 性能监控

应用性能监控:

代码块

```
1  import { createPrometheusMetrics } from 'prom-client';
2
3  const httpRequestDuration = new createPrometheusMetrics.Histogram({
4    name: 'http_request_duration_seconds',
5    help: 'Duration of HTTP requests in seconds',
6    labelNames: ['method', 'route', 'status']
7  });
8
9  const activeConnections = new createPrometheusMetrics.Gauge({
10    name: 'websocket_active_connections',
11    help: 'Number of active WebSocket connections'
12  });
```

日志聚合:

代码块

```
1  import logging
2  from pythonjsonlogger import jsonlogger
3
4  logger = logging.getLogger()
```

```
5 logger.setLevel(logging.INFO)
6
7 json_handler = logging.StreamHandler()
8 formatter = jsonlogger.JsonFormatter()
9 json_handler.setFormatter(formatter)
10 logger.addHandler(json_handler)
```

7.2 安全措施

API安全:

代码块

```
1 import helmet from 'helmet';
2 import rateLimit from 'express-rate-limit';
3
4 // 安全头设置
5 app.use(helmet({
6   contentSecurityPolicy: {
7     directives: {
8       defaultSrc: ['self'],
9       styleSrc: ['self', 'unsafe-inline'],
10      scriptSrc: ['self'],
11      imgSrc: ['self', 'data:', 'https:']
12    }
13  }
14 }));
15
16 // 速率限制
17 const limiter = rateLimit({
18   windowMs: 15 * 60 * 1000, // 15分钟
19   max: 100, // 限制每IP 100次请求
20   message: 'Too many requests from this IP'
21 });
22
23 app.use('/api/', limiter);
```

数据加密:

代码块

```
1 from cryptography.fernet import Fernet
2
3 class DataEncryption:
4     def __init__(self):
5         self.key = os.getenv("ENCRYPTION_KEY").encode()
```

```
6         self.cipher = Fernet(self.key)
7
8     def encrypt_sensitive_data(self, data: str) -> str:
9         encrypted_data = self.cipher.encrypt(data.encode())
10        return encrypted_data.decode()
11
12    def decrypt_sensitive_data(self, encrypted_data: str) -> str:
13        decrypted_data = self.cipher.decrypt(encrypted_data.encode())
14        return decrypted_data.decode()
```

文档版本: v1.0

创建日期: 2026年1月16日