**Sistema IA Guidato da Knowledge Base Personalizzata**

**1. ARCHITETTURA KNOWLEDGE BASE STRUTTURATA**

**1.1 Struttura della KB per Contesti**

knowledge\_base:

protocolli\_clinici:

lombalgia:

valutazione\_iniziale:

- red\_flags: ["dolore notturno costante", "perdita peso", "febbre"]

- yellow\_flags: ["paura movimento", "catastrofizzazione", "evitamento"]

- test\_essenziali: ["Lasègue", "Wasserman", "SLUMP test"]

- scale\_valutazione: ["Oswestry", "Roland Morris", "VAS"]

protocollo\_trattamento:

fase\_acuta:

- tecniche: ["mobilizzazione grado I-II", "TENS", "educazione"]

- controindicazioni: ["manipolazioni", "esercizi intensi"]

- durata: "1-2 settimane"

- frequenza: "2-3 sedute/settimana"

fase\_subacuta:

- tecniche: ["mobilizzazione grado III-IV", "esercizio terapeutico"]

- progressione: "graduale carico"

- obiettivi: ["riduzione dolore 50%", "ROM 80%"]

criteri\_dimissione:

- "VAS < 3"

- "Oswestry < 20%"

- "Obiettivi funzionali raggiunti"

- "Autogestione appresa"

spalla:

sindrome\_impingement:

valutazione: [...]

trattamento: [...]

capsulite\_adesiva:

valutazione: [...]

trattamento: [...]

ginocchio:

post\_lca:

protocollo\_riabilitativo: [...]

criteri\_progressione: [...]

linee\_guida\_comunicazione:

primo\_contatto:

- tono: "empatico e professionale"

- informazioni\_essenziali: ["orari", "documenti necessari", "abbigliamento"]

- rassicurazioni: ["normale sentire preoccupazione", "siamo qui per aiutare"]

spiegazione\_diagnosi:

- linguaggio: "semplice, no tecnicismi eccessivi"

- metafore\_utili:

disco: "cuscinetto ammortizzatore"

infiammazione: "sistema allarme del corpo"

- sempre\_includere: ["prognosi", "tempi recupero", "cosa può fare paziente"]

gestione\_aspettative:

- realistiche: "miglioramento graduale, non lineare"

- positive: "la maggior parte dei pazienti migliora"

- empowerment: "lei è parte attiva del recupero"

faq\_personalizzate:

dolore\_post\_seduta:

risposta\_standard: "È normale avvertire indolenzimento nelle 24-48h..."

quando\_preoccuparsi: "Se dolore intenso oltre 48h o sintomi neurologici"

cosa\_fare: ["ghiaccio 15min", "esercizi leggeri", "contattarci se persiste"]

numero\_sedute:

fattori: ["cronicità", "età", "compliance", "complessità"]

media\_per\_patologia:

lombalgia\_acuta: "4-6 sedute"

lombalgia\_cronica: "8-12 sedute"

post\_chirurgico: "15-20 sedute"

**1.2 Database Vettoriale per Ricerca Semantica**

class KnowledgeBaseVectorDB:

"""

Sistema di ricerca semantica nella KB usando embeddings

"""

def \_\_init\_\_(self):

self.vector\_store = ChromaDB() # o Pinecone, Weaviate

self.embedder = EmbeddingModel('multilingual-e5-large')

def index\_knowledge(self):

"""Indicizza tutta la KB in vettori"""

documents = []

# Protocolli clinici

for patologia, protocollo in self.kb.protocolli\_clinici.items():

doc = {

'text': json.dumps(protocollo),

'metadata': {

'tipo': 'protocollo\_clinico',

'patologia': patologia,

'tags': self.extract\_tags(protocollo)

}

}

documents.append(doc)

# Casi clinici storici (anonimizzati)

for caso in self.kb.casi\_studio:

doc = {

'text': caso['descrizione'],

'metadata': {

'tipo': 'caso\_clinico',

'outcome': caso['risultato'],

'durata\_trattamento': caso['sedute']

}

}

documents.append(doc)

# Embed e salva

embeddings = self.embedder.encode([d['text'] for d in documents])

self.vector\_store.add(embeddings, documents)

def search\_similar(self, query, context\_type=None, k=5):

"""Cerca contenuti simili nella KB"""

query\_embedding = self.embedder.encode(query)

filters = {}

if context\_type:

filters['tipo'] = context\_type

results = self.vector\_store.similarity\_search(

query\_embedding,

k=k,

filter=filters

)

return self.rank\_by\_relevance(results, query)

**2. MODULI IA ASSISTITI PER CONTESTO**

**2.1 Assistente Valutazione Clinica**

class ClinicalAssessmentAI {

constructor(knowledgeBase) {

this.kb = knowledgeBase;

this.model = 'gpt-4-medical-tuned'; // Fine-tuned su dati medici

}

async suggestAssessment(patientData) {

// Recupera protocolli simili dalla KB

const similarCases = await this.kb.findSimilarCases({

symptoms: patientData.symptoms,

age: patientData.age,

history: patientData.medicalHistory

});

// Genera suggerimenti basati su KB

const suggestions = {

test\_consigliati: [],

red\_flags\_check: [],

scale\_valutative: [],

diagnosi\_differenziali: []

};

// Analizza sintomi per red flags

for (const symptom of patientData.symptoms) {

const redFlags = await this.kb.checkRedFlags(symptom);

if (redFlags.length > 0) {

suggestions.red\_flags\_check.push({

sintomo: symptom,

flags: redFlags,

urgenza: this.calculateUrgency(redFlags),

azione: this.kb.getProtocolAction(redFlags)

});

}

}

// Suggerisci test basati su pattern

const testPattern = await this.analyzeTestPattern(patientData, similarCases);

suggestions.test\_consigliati = testPattern.tests.map(test => ({

nome: test,

razionale: this.kb.getTestRationale(test, patientData),

priorita: this.kb.getTestPriority(test, patientData)

}));

// Scale valutative appropriate

suggestions.scale\_valutative = this.kb.getRecommendedScales(

patientData.distretto,

patientData.patologia\_sospetta

);

return suggestions;

}

async generateClinicalReasoning(valutazione) {

"""

Genera ragionamento clinico guidato da KB

"""

const reasoning = {

analisi\_sintomi: {},

ipotesi\_diagnostiche: [],

test\_confermatori: [],

piano\_trattamento\_suggerito: {}

};

// Usa la KB per strutturare il ragionamento

const template = this.kb.getReasoningTemplate(valutazione.tipo);

// Popola con IA guidata

reasoning.analisi\_sintomi = await this.analyzeSymptoms(

valutazione.sintomi,

template.symptom\_analysis

);

reasoning.ipotesi\_diagnostiche = await this.generateHypotheses(

valutazione,

template.differential\_diagnosis

);

return reasoning;

}

}

**2.2 Assistente Generazione Referti**

class ReportGenerationAI:

def \_\_init\_\_(self, knowledge\_base):

self.kb = knowledge\_base

self.templates = self.kb.load\_report\_templates()

self.medical\_nlp = MedicalNLP()

def generate\_physiotherapy\_report(self, session\_data, patient\_history):

"""

Genera referto fisioterapico basato su KB e dati sessione

"""

# Recupera template appropriato dalla KB

template = self.kb.get\_template(

pathology=session\_data['diagnosis'],

report\_type='progress\_report'

)

# Analizza progressi usando KB

progress\_analysis = self.analyze\_progress(

current=session\_data,

history=patient\_history,

expected=self.kb.get\_expected\_progress(session\_data['diagnosis'])

)

# Genera testo del referto

report = {

'valutazione\_oggettiva': self.generate\_objective\_assessment(

session\_data,

template['objective\_structure']

),

'analisi\_progressi': self.format\_progress\_analysis(

progress\_analysis,

self.kb.get\_progress\_descriptors()

),

'piano\_trattamento': self.suggest\_treatment\_modifications(

progress\_analysis,

self.kb.get\_treatment\_protocols(session\_data['diagnosis'])

),

'raccomandazioni': self.generate\_recommendations(

patient\_data=session\_data,

guidelines=self.kb.get\_home\_exercise\_guidelines()

),

'prognosi': self.estimate\_prognosis(

progress\_rate=progress\_analysis['rate'],

similar\_cases=self.kb.find\_similar\_outcomes()

)

}

# Validazione con KB

report = self.validate\_against\_guidelines(report)

return self.format\_report(report, template)

def auto\_complete\_from\_notes(self, therapist\_notes):

"""

Completa automaticamente campi del referto da note libere

"""

# Estrai entità mediche

entities = self.medical\_nlp.extract\_entities(therapist\_notes)

# Mappa su struttura referto usando KB

structured\_data = {}

for entity in entities:

field = self.kb.map\_entity\_to\_field(entity)

if field:

structured\_data[field] = {

'value': entity.value,

'confidence': entity.confidence,

'source': entity.text\_span

}

# Suggerisci completamenti basati su pattern

suggestions = self.kb.suggest\_completions(structured\_data)

return {

'extracted': structured\_data,

'suggestions': suggestions

}

**2.3 Chatbot Assistente Paziente**

class PatientAssistantAI {

constructor(knowledgeBase) {

this.kb = knowledgeBase;

this.contextMemory = new Map();

this.emotionalTone = new EmotionalAnalyzer();

}

async handlePatientQuery(query, patientId) {

// Recupera contesto paziente

const context = await this.loadPatientContext(patientId);

// Analizza intento e sentimento

const intent = await this.detectIntent(query);

const emotion = this.emotionalTone.analyze(query);

// Cerca risposta nella KB

const kbResponse = await this.kb.searchFAQ(query, {

patientCondition: context.diagnosis,

treatmentPhase: context.currentPhase

});

// Genera risposta personalizzata

let response;

switch(intent.type) {

case 'pain\_concern':

response = await this.handlePainConcern(

query,

context,

this.kb.getPainManagementGuidelines()

);

break;

case 'exercise\_question':

response = await this.handleExerciseQuery(

query,

context,

this.kb.getExerciseDatabase()

);

break;

case 'appointment\_request':

response = await this.handleAppointmentRequest(

query,

context,

await this.checkAvailability()

);

break;

case 'progress\_inquiry':

response = await this.explainProgress(

context.progressData,

this.kb.getProgressExplanations()

);

break;

default:

response = await this.generateGeneralResponse(

query,

kbResponse,

context

);

}

// Adatta tono in base a stato emotivo

response = this.adaptTone(response, emotion);

// Aggiungi risorse dalla KB se pertinenti

response.resources = this.kb.getRelevantResources(intent, context);

// Salva contesto conversazione

this.updateContext(patientId, {query, response, intent, emotion});

return response;

}

async generateExerciseInstructions(exerciseId, patientProfile) {

"""

Genera istruzioni personalizzate per esercizi

"""

const exercise = this.kb.getExercise(exerciseId);

const modifications = this.kb.getModifications(exercise, patientProfile);

return {

nome: exercise.name,

descrizione: this.personalizeDescription(

exercise.description,

patientProfile.comprehensionLevel

),

istruzioni\_passo\_passo: exercise.steps.map(step =>

this.adaptInstruction(step, patientProfile.limitations)

),

modifiche\_consigliate: modifications.map(mod => ({

motivo: mod.reason,

modifica: mod.description,

quando\_applicare: mod.condition

})),

errori\_comuni: this.kb.getCommonMistakes(exerciseId).map(mistake => ({

errore: mistake.description,

correzione: mistake.correction,

conseguenze: mistake.risks

})),

progressione: this.kb.getProgressionPath(

exerciseId,

patientProfile.currentLevel

),

video\_tutorial: this.kb.getVideoLink(exerciseId),

quando\_contattare: this.kb.getWarningSignsExercise(exerciseId)

};

}

}

**2.4 Sistema Predittivo Outcome**

class OutcomePredictionAI:

def \_\_init\_\_(self, knowledge\_base):

self.kb = knowledge\_base

self.ml\_models = self.load\_trained\_models()

def predict\_treatment\_outcome(self, patient\_data):

"""

Predice outcome basandosi su casi simili nella KB

"""

# Trova casi simili nella KB

similar\_cases = self.kb.find\_similar\_patients(

age=patient\_data['age'],

diagnosis=patient\_data['diagnosis'],

severity=patient\_data['severity'],

comorbidities=patient\_data['comorbidities'],

n\_cases=100

)

# Feature engineering basato su KB

features = self.extract\_predictive\_features(

patient\_data,

self.kb.get\_predictive\_factors()

)

# Ensemble prediction

predictions = {}

# Modello 1: Random Forest su casi storici

predictions['rf'] = self.ml\_models['random\_forest'].predict(features)

# Modello 2: Neural Network

predictions['nn'] = self.ml\_models['neural\_net'].predict(features)

# Modello 3: Similarity-based

predictions['similarity'] = self.similarity\_based\_prediction(

patient\_data,

similar\_cases

)

# Combina predizioni con pesi dalla KB

weights = self.kb.get\_model\_weights(patient\_data['diagnosis'])

final\_prediction = self.weighted\_ensemble(predictions, weights)

return {

'predicted\_sessions': final\_prediction['n\_sessions'],

'success\_probability': final\_prediction['success\_rate'],

'recovery\_timeline': self.generate\_timeline(final\_prediction),

'confidence': self.calculate\_confidence(predictions),

'similar\_cases\_summary': self.summarize\_similar\_cases(similar\_cases),

'risk\_factors': self.identify\_risk\_factors(patient\_data, similar\_cases),

'optimization\_suggestions': self.suggest\_optimizations(final\_prediction)

}

def predict\_dropout\_risk(self, patient\_id):

"""

Predice rischio abbandono terapia

"""

patient = self.get\_patient\_data(patient\_id)

risk\_factors = {

'compliance': self.analyze\_compliance\_pattern(patient),

'progress\_rate': self.analyze\_progress\_rate(patient),

'satisfaction': self.analyze\_satisfaction\_trend(patient),

'external\_factors': self.kb.get\_dropout\_factors()

}

risk\_score = self.calculate\_dropout\_risk(risk\_factors)

if risk\_score > 0.7:

return {

'risk\_level': 'HIGH',

'score': risk\_score,

'main\_factors': self.get\_top\_risk\_factors(risk\_factors),

'interventions': self.kb.get\_retention\_strategies(risk\_factors),

'suggested\_actions': self.generate\_retention\_plan(patient, risk\_factors)

}

return {'risk\_level': 'LOW', 'score': risk\_score}

**3. SISTEMA DI APPRENDIMENTO CONTINUO**

**3.1 Feedback Loop per Miglioramento KB**

class KnowledgeBaseEvolution:

def \_\_init\_\_(self):

self.kb = KnowledgeBase()

self.learning\_queue = Queue()

self.validation\_threshold = 0.8

def learn\_from\_outcomes(self, treatment\_id):

"""

Apprende da outcome reali per aggiornare KB

"""

outcome = self.get\_treatment\_outcome(treatment\_id)

prediction = self.get\_initial\_prediction(treatment\_id)

# Calcola delta

learning\_data = {

'prediction\_error': abs(outcome['actual'] - prediction['predicted']),

'unexpected\_factors': self.identify\_unexpected\_factors(outcome),

'successful\_techniques': outcome['effective\_interventions'],

'ineffective\_techniques': outcome['ineffective\_interventions']

}

# Aggiungi a coda apprendimento

self.learning\_queue.add(learning\_data)

# Se abbastanza dati, aggiorna KB

if self.learning\_queue.size() >= 50:

self.update\_knowledge\_base()

def update\_knowledge\_base(self):

"""

Aggiorna KB con nuove conoscenze validate

"""

learnings = self.learning\_queue.get\_all()

# Aggrega pattern

patterns = self.extract\_patterns(learnings)

# Valida con esperti

validated\_patterns = []

for pattern in patterns:

if pattern['confidence'] > self.validation\_threshold:

validated\_patterns.append(pattern)

else:

# Invia a esperti per revisione

self.send\_for\_expert\_validation(pattern)

# Aggiorna KB

for pattern in validated\_patterns:

self.kb.update\_protocol(

condition=pattern['condition'],

updates=pattern['protocol\_updates']

)

self.kb.add\_case\_study({

'pattern': pattern,

'evidence': pattern['supporting\_cases'],

'timestamp': datetime.now()

})

# Retrain modelli predittivi

self.retrain\_models()

**3.2 Personalizzazione per Terapista**

class TherapistAIPersonalization {

constructor(therapistId) {

this.therapistId = therapistId;

this.preferences = this.loadPreferences();

this.writingStyle = this.analyzeWritingStyle();

this.commonProtocols = this.loadCommonProtocols();

}

personalizeAISuggestions(baseSuggestions) {

// Adatta suggerimenti allo stile del terapista

return {

...baseSuggestions,

// Usa terminologia preferita

terminology: this.translateToPreferredTerms(baseSuggestions),

// Ordina per protocolli più usati

protocols: this.rankByUsagePattern(

baseSuggestions.protocols,

this.commonProtocols

),

// Applica modifiche tipiche del terapista

modifications: this.applyTypicalModifications(

baseSuggestions.treatment,

this.preferences.typical\_adaptations

),

// Formatta secondo stile di documentazione

formatting: this.applyFormattingStyle(

baseSuggestions,

this.writingStyle

)

};

}

learnFromTherapistChoices(suggestion, actualChoice) {

// Apprendi dalle scelte del terapista

const delta = this.calculateDelta(suggestion, actualChoice);

// Aggiorna preferenze

this.preferences.update(delta);

// Aggiorna pattern comuni

if (actualChoice.protocol) {

this.commonProtocols.increment(actualChoice.protocol);

}

// Aggiorna stile scrittura

if (actualChoice.notes) {

this.writingStyle.addSample(actualChoice.notes);

}

// Salva per personalizzazione futura

this.save();

}

}

**4. INTERFACCIA UTENTE IA**

**4.1 Dashboard IA Insights**

const AIDashboard = {

widgets: {

predictionAccuracy: {

title: "Accuratezza Predizioni IA",

current: "87%",

trend: "+3%",

details: {

outcome\_predictions: "89%",

dropout\_predictions: "84%",

session\_estimates: "86%"

}

},

suggestionsAdopted: {

title: "Suggerimenti IA Adottati",

adoption\_rate: "72%",

most\_adopted: [

"Test valutativi (85%)",

"Modifiche esercizi (78%)",

"Progressioni (71%)"

],

least\_adopted: [

"Cambi protocollo (45%)",

"Tecniche alternative (52%)"

]

},

anomalyDetection: {

title: "Anomalie Rilevate",

this\_week: 3,

alerts: [

{

type: "pattern\_insolito",

description: "Recupero più lento del previsto - 5 casi",

action: "Review protocollo lombalgia cronica"

},

{

type: "dropout\_risk",

description: "3 pazienti ad alto rischio abbandono",

action: "Contatto preventivo consigliato"

}

]

},

knowledgeBaseStats: {

title: "Knowledge Base",

total\_entries: "12,456",

last\_update: "2 ore fa",

new\_patterns: 15,

pending\_validation: 3

}

}

};

**4.2 Assistente IA in Tempo Reale**

class RealTimeAIAssistant:

def \_\_init\_\_(self):

self.kb = KnowledgeBase()

self.context\_tracker = ContextTracker()

def provide\_real\_time\_suggestions(self, current\_action, context):

"""

Fornisce suggerimenti mentre il terapista lavora

"""

suggestions = []

if current\_action == 'valutazione\_rom':

# Suggerisci test complementari

suggestions.append({

'tipo': 'test\_aggiuntivo',

'suggerimento': f"Basato su ROM limitato, considera {self.kb.suggest\_tests(context)}",

'razionale': self.kb.get\_test\_rationale(context)

})

elif current\_action == 'scrittura\_referto':

# Auto-completamento intelligente

text = context['partial\_text']

suggestions.append({

'tipo': 'completamento',

'suggerimenti': self.kb.autocomplete\_medical\_text(text),

'template\_relevanti': self.kb.get\_relevant\_templates(text)

})

elif current\_action == 'pianificazione\_trattamento':

# Suggerisci basandosi su outcome simili

similar\_success = self.kb.find\_successful\_treatments(context)

suggestions.append({

'tipo': 'piano\_trattamento',

'opzioni': similar\_success[:3],

'evidenza': self.kb.get\_evidence\_level(similar\_success)

})

return self.rank\_suggestions(suggestions, context)

def generate\_session\_summary(self, session\_data):

"""

Genera sommario automatico della seduta

"""

summary = {

'obiettivi\_raggiunti': self.evaluate\_goals(

session\_data['goals'],

session\_data['outcomes']

),

'progressi\_chiave': self.extract\_key\_progress(

session\_data,

self.kb.get\_progress\_markers()

),

'note\_cliniche': self.generate\_clinical\_notes(

session\_data,

self.kb.get\_note\_template()

),

'prossimi\_passi': self.suggest\_next\_steps(

session\_data,

self.kb.get\_progression\_protocols()

),

'homework\_paziente': self.generate\_homework(

session\_data['patient\_profile'],

session\_data['current\_status']

)

}

return summary

**5. CONFIGURAZIONE E TRAINING IA**

**5.1 Sistema di Training Modelli**

class AIModelTraining:

def \_\_init\_\_(self):

self.training\_data = TrainingDataManager()

self.model\_registry = ModelRegistry()

def prepare\_training\_data(self):

"""

Prepara dati da KB e storico per training

"""

# Estrai da database

historical\_data = self.extract\_historical\_cases()

# Anonimizza

anonymized = self.anonymize\_patient\_data(historical\_data)

# Augmentation con variazioni

augmented = self.augment\_data(anonymized)

# Split train/val/test

return self.split\_data(augmented)

def train\_specialized\_models(self):

"""

Training modelli specializzati per contesto

"""

models = {}

# Modello predizione outcome

models['outcome\_predictor'] = self.train\_outcome\_model()

# Modello classificazione urgenza

models['urgency\_classifier'] = self.train\_urgency\_model()

# Modello suggerimento esercizi

models['exercise\_recommender'] = self.train\_exercise\_model()

# Modello analisi progresso

models['progress\_analyzer'] = self.train\_progress\_model()

# Modello generazione testo

models['text\_generator'] = self.finetune\_language\_model()

return models

def continuous\_learning\_pipeline(self):

"""

Pipeline per apprendimento continuo

"""

while True:

# Raccogli nuovi dati

new\_data = self.collect\_new\_data()

if len(new\_data) >= self.batch\_size:

# Valida qualità dati

validated = self.validate\_data\_quality(new\_data)

# Aggiorna modelli incrementalmente

for model\_name, model in self.models.items():

model.incremental\_update(validated)

# Test performance

metrics = self.evaluate\_models()

# Rollback se performance degrada

if metrics['degradation'] > self.threshold:

self.rollback\_models()

else:

self.commit\_model\_updates()

time.sleep(self.update\_interval)

**5.2 Configurazione per Contesto Clinico**

ai\_configuration:

contexts:

valutazione\_iniziale:

models:

- red\_flag\_detector

- differential\_diagnosis

- test\_recommender

kb\_sections:

- protocolli\_valutazione

- test\_clinici

- red\_flags\_database

confidence\_threshold: 0.85

require\_validation: true

trattamento:

models:

- technique\_selector

- progression\_advisor

- modification\_suggester

kb\_sections:

- tecniche\_manuali

- esercizi\_terapeutici

- progressioni\_standard

personalization: high

adapt\_to\_therapist: true

refertazione:

models:

- text\_generator

- medical\_encoder

- outcome\_summarizer

kb\_sections:

- template\_referti

- terminologia\_medica

- linee\_guida\_documentazione

language: italian

style: professional\_medical

comunicazione\_paziente:

models:

- intent\_classifier

- emotion\_detector

- response\_generator

kb\_sections:

- faq

- spiegazioni\_semplificate

- consigli\_autogestione

tone: empathetic

complexity: adaptive

**6. PRIVACY E SICUREZZA IA**

**6.1 Gestione Dati Sensibili**

class AIPrivacyManager:

def \_\_init\_\_(self):

self.encryption = AESEncryption()

self.anonymizer = DataAnonymizer()

def process\_for\_ai(self, patient\_data):

"""

Prepara dati per elaborazione IA preservando privacy

"""

# Rimuovi identificatori diretti

anonymized = self.anonymizer.remove\_identifiers(patient\_data)

# Generalizza dati sensibili

generalized = self.generalize\_sensitive\_data(anonymized)

# Aggiungi rumore differenziale per privacy

private\_data = self.add\_differential\_privacy(generalized)

# Cripta per trasferimento

encrypted = self.encryption.encrypt(private\_data)

return encrypted

def federated\_learning\_setup(self):

"""

Setup per apprendimento federato (dati restano locali)

"""

return {

'local\_training': True,

'model\_aggregation': 'secure\_aggregation',

'data\_sharing': 'never',

'gradient\_sharing': 'encrypted\_only'

}

**6.2 Audit Trail IA**

CREATE TABLE ai\_decisions\_audit (

id SERIAL PRIMARY KEY,

timestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

model\_name VARCHAR(100),

model\_version VARCHAR(50),

input\_hash VARCHAR(256), -- Hash dei dati input

output JSON,

confidence\_score DECIMAL(3,2),

therapist\_id INTEGER,

patient\_id\_hash VARCHAR(256), -- Hash per privacy

decision\_accepted BOOLEAN,

modification\_made TEXT,

context\_type VARCHAR(50),

kb\_version VARCHAR(50)

);

CREATE TABLE ai\_performance\_metrics (

id SERIAL PRIMARY KEY,

date DATE,

model\_name VARCHAR(100),

accuracy DECIMAL(5,2),

precision DECIMAL(5,2),

recall DECIMAL(5,2),

f1\_score DECIMAL(5,2),

adoption\_rate DECIMAL(5,2),

false\_positive\_rate DECIMAL(5,2),

false\_negative\_rate DECIMAL(5,2),

user\_satisfaction DECIMAL(3,2)

);

**7. MONITORAGGIO E METRICHE**

**7.1 Dashboard Performance IA**

const AIPerformanceDashboard = {

renderMetrics() {

return {

accuracy\_by\_context: {

valutazione: "89%",

trattamento: "85%",

prognosi: "78%",

comunicazione: "92%"

},

time\_saved: {

refertazione: "65% riduzione tempo",

valutazione: "40% più efficiente",

pianificazione: "55% più veloce"

},

clinical\_outcomes: {

improvement\_rate: "+12% con IA",

dropout\_reduction: "-23% abbandoni",

patient\_satisfaction: "+18% soddisfazione"

},

kb\_growth: {

entries\_added: "156 questo mese",

patterns\_discovered: "23 nuovi",

validations\_pending: 8

},

cost\_benefit: {

roi: "320%",

cost\_per\_prediction: "€0.02",

value\_generated: "€45,000/anno"

}

};

}

};