Capitolo 4: Scelte Tecnologiche

Questo capitolo presenta le principali scelte tecnologiche adottate nello sviluppo del chatbot AI (sistema automatizzato di conversazione basato su Intelligenza Artificiale) per il supporto tecnico, con particolare attenzione agli strumenti e framework che hanno permesso l'integrazione tra intelligenza artificiale e sistemi aziendali esistenti.

4.1 OpenAI GPT-4: Cuore dell'Intelligenza Conversazionale

La scelta di OpenAI GPT-4 (Generative Pre-trained Transformer 4, modello di intelligenza artificiale avanzato sviluppato dall'azienda OpenAI) come motore di intelligenza artificiale è stata determinata dalla sua capacità di gestire contenuti tecnici complessi e dalla possibilità di estendere le sue funzionalità attraverso API (Application Programming Interface, interfacce che permettono la comunicazione tra diversi sistemi software).

Come evidenziato nel file `openai-handlers.ts` (componente del codice che gestisce le comunicazioni con il servizio OpenAI), l'implementazione sfrutta tre funzionalità chiave dell'API OpenAI:

1. **Function Calling**: Meccanismo che consente al modello di intelligenza artificiale di riconoscere quando è necessario eseguire operazioni concrete sulla piattaforma proprietaria Konsolex (sistema operativo cloud unificato sviluppato da OnTheCloud).

```
// Esempio dall'implementazione in openai-handlers.ts
async function handleToolCalls(openai: OpenAI, run: any, threadId: string,
runId: string) {
  if (run.required_action?.type === 'submit_tool_outputs') {
    const toolCalls = run.required_action.submit_tool_outputs.tool_calls;
    // Esecuzione di funzioni basate sulla richiesta dell'AI
  }
}
```

2. **Thread Management** (Gestione delle conversazioni): Sistema che permette il mantenimento del contesto conversazionale, consentendo all'intelligenza artificiale di OpenAI di conservare la continuità del dialogo con il cliente attraverso più interazioni.

```
// Gestione e reset dei thread in openai-handlers.ts
const thread = await openai.beta.threads.create();
await userOps.updateThreadByUserId(userId, thread.id);
```

3. **Parametrizzazione Avanzata**: La qualità e coerenza delle risposte dell'assistente AI sono state calibrate attraverso parametri specifici, in particolare "temperature" (parametro che controlla la casualità delle risposte) e "top_p" (parametro che influenza la distribuzione di probabilità delle parole generate), che influenzano direttamente il processo di generazione del testo:

```
run = await openai.beta.threads.runs.create(
  user.thread_id,
  {
    assistant_id: user.assistant_id,
    tools: openaiTools,
    temperature: 0.1,
    top_p: 1
  }
);
```

L'implementazione adotta deliberatamente un approccio deterministico (temperature: 0.1) che riduce significativamente la casualità nelle risposte generate. Questa configurazione garantisce che:

- Le soluzioni tecniche fornite siano coerenti tra interazioni simili
- I processi di risoluzione dei problemi seguano percorsi standardizzati
- Le istruzioni operative fornite rispettino rigorosamente le procedure documentate

4.2 Stack tecnologico preesistente: TypeScript, Node.js, Express

L'architettura tecnologica del progetto è stata sviluppata all'interno di un insieme di tecnologie interconnesse, definito dalle scelte tecnologiche già consolidate dalla società OnTheCloud. L'integrazione del chatbot ha richiesto l'adattamento alle tecnologie aziendali standardizzate, garantendo coerenza con l'ecosistema applicativo esistente e facilitando la manutenzione futura.

Il progetto è stato sviluppato in TypeScript (linguaggio di programmazione che estende JavaScript aggiungendo definizioni di tipo statico), linguaggio standard nell'ecosistema aziendale, beneficiando di caratteristiche come:

- **Tipizzazione statica**: Integrazione sicura con i sistemi esistenti grazie ai controlli di tipo che prevengono errori durante lo sviluppo
- **Intellisense e autocompletamento**: Accelerazione della curva di apprendimento delle API aziendali attraverso suggerimenti automatici nell'ambiente di sviluppo
- **Documentazione implicita**: Comprensione più rapida delle interfacce preesistenti grazie alle definizioni di tipo

L'implementazione ha rispettato il pattern di interfacce tipizzate già in uso per la gestione di entità come messaggi e utenti:

```
interface SendMessageRequest {
  text: string;
  userId: string;
  tg_id: string;
}
```

4.2.2 Node.js ed Express.js

Node.js (ambiente di esecuzione JavaScript lato server) è stato scelto come runtime per la sua eccellente gestione delle operazioni I/O non bloccanti, cruciale per un sistema di chat reattivo che deve gestire diverse conversazioni simultanee.

Express.js (framework web per Node.js che semplifica lo sviluppo di applicazioni web) ha fornito un framework leggero ma potente per l'implementazione delle API REST (interfacce web standardizzate), come evidente nel file `web-server.ts`:

```
app.post(API.SEND_MESSAGE, async (req, res) => {
  const { text, userId, tg_id } = req.body as SendMessageRequest;
  // Gestione della richiesta e generazione risposta
});
```

4.3 Database e ORM Sequelize

Per la persistenza dei dati (salvataggio permanente delle informazioni) è stato adottato un approccio relazionale gestito attraverso Sequelize ORM (Object-Relational Mapping, libreria che facilita l'interazione con database relazionali attraverso un'interfaccia orientata agli oggetti), che ha fornito:

1. **Astrazione del database**: Indipendenza dal DBMS (Database Management System) specifico sottostante
2. **Modelli tipizzati**: Integrazione naturale con TypeScript
3. **Query builder**: Semplificazione delle operazioni CRUD (Create, Read, Update, Delete - operazioni fondamentali sui dati)

I repository come `message-repository.ts` e `user-repository.ts` implementano pattern di accesso ai dati ben strutturati:

```
// Esempio di pattern repository con Sequelize
export async function findByUserId(userId: string): Promise<User | null> {
  return await User.findOne({ where: { user_id: userId } });
}
```

Le entità principali gestite includono:

- **Users**: Profili utente con riferimenti a thread OpenAI
- **Messages**: Conversazioni con tipologia e metadati associati
- **Assistants**: Configurazioni degli assistenti virtuali
- **Score**: Risultati delle risposte in relazione alla domanda posta dal client, implementato per futuri utilizzi.

4.4 Integrazione con API esterne

4.4.1 Telegram Bot API

L'integrazione con Telegram (piattaforma di messaggistica istantanea cloud-based) è stata implementata tramite la libreria Telegraf (framework che semplifica lo sviluppo di bot per Telegram), fornendo un'interfaccia utente accessibile e familiare.

Il file `bot.ts` gestisce i comandi e le interazioni, mentre `endpoint.ts` implementa funzionalità come l'invio di notifiche ticket agli amministratori:

```
await bot.telegram.sendMessage(adminUserId, message, {
  reply_markup: inlineKeyboard,
});
```

4.4.2 Konsolex API

L'integrazione con la piattaforma proprietaria Konsolex rappresenta un elemento distintivo del progetto. Il file `endpoint.ts` implementa numerose funzioni che interagiscono con l'API Konsolex:

```
export async function restartServer(userId: string, serverName: string):
Promise<boolean> {
  try {
    const server_id = await getServerId(userId, serverName);
    const response = await axios.post(
      KONSOLEX_ENDPOINT.RESTART_SERVER,
      { id: server_id.id },
      {
        headers: {
          'Accept': 'application/json',
          'Content-Type': 'application/json',
          'userid': userId
        }
      }
    );
    return true;
  } catch (error) {
    console.error("Errore nel riavvio del server:", error);
    return false;
  }
}
```

Le principali categorie di API utilizzate includono:

- **Gestione utenti**: Autenticazione e sincronizzazione profili
- **Operazioni server**: Riavvio, monitoraggio e configurazione
- **Gestione domini**: Verifica disponibilità, configurazione DNS (sistema che traduce i nomi di dominio in indirizzi IP)
- **Container management**: Restart e scaling (adattamento dinamico delle risorse) di risorse containerizzate (tecnologia che impacchetta applicazioni e le loro dipendenze)

La centralizzazione degli endpoint nel file `constants.ts` garantisce consistenza nelle interazioni con le API esterne e semplifica eventuali aggiornamenti futuri.

4.5 Considerazioni finali

L'architettura tecnologica adottata bilancia efficacemente innovazione (GPT-4) e solidità (TypeScript, Node.js), creando un sistema capace di gestire richieste tecniche complesse attraverso un'interfaccia conversazionale intuitiva. L'integrazione con i sistemi esistenti di OnTheCloud è stata garantita attraverso API ben documentate e un design modulare che facilita future evoluzioni del sistema.