Backend Migration Plan: Node.js to Python

Overview

This document outlines a comprehensive plan for migrating the existing Node.js/Express backend to Python. The migration is particularly motivated by better integration with the Model Context Protocol (MCP) for the co-pilot application.

Rationale for Migration

Why Migrate to Python?

1. **Superior MCP Integration**: Python offers better integration with MCP through its official SDK
2. **Rich Ecosystem**: Python's ecosystem for AI/ML integration is more mature
3. **Single Language Stack**: Using Python for both MCP server and backend simplifies maintenance
4. **Performance**: For compute-intensive operations like processing terminal commands, Python offers strong performance

Migration Strategy

High-Level Approach

We'll adopt a phased, incremental migration strategy:

1. **Assessment Phase**: Catalog all existing functionality and endpoints
2. **Parallel Development**: Build the Python backend alongside the existing Node.js backend
3. **Component Migration**: Migrate components one by one, starting with non-critical features
4. **Testing & Verification**: Comprehensive testing of each migrated component
5. **Gradual Cutover**: Switch traffic gradually from old to new system
6. **Decommissioning**: Remove the old Node.js backend once migration is complete

Technology Selection

Recommended Python Stack

Component	Node.js Current	Python Replacement	Justification
Web Framework	Express.js	FastAPI	Modern, async-first, excellent type hints, auto-documentation
Database ORM	Native SQL/Better-SQLite3	SQLAlchemy	Industry standard ORM with excellent SQLite support
Authentication	express-session	FastAPI + JWT + Pydantic	Modern token-based auth with strong validation

Component	Node.js Current	Python Replacement	Justification
API Documentation	None/Manual	FastAPI/OpenAPI	Automatic API documentation
MCP Integration	TypeScript SDK	Python SDK	Native Python SDK from MCP
WebSockets	ws/Socket.io	FastAPI + WebSockets	Native async WebSocket support
Process Management	PM2	Gunicorn + Uvicorn	Production-grade ASGI servers

Detailed Migration Plan

1. Database Migration

Current Database Schema

The current application uses SQLite with direct queries through Better-SQLite3.

Migration Steps

1. **Schema Analysis**:
   - Map all existing tables, relationships, and constraints
   - Document all SQL queries used in the application
2. **SQLAlchemy Model Creation**:

```
# Example SQLAlchemy models
from sqlalchemy import Column, Integer, String, ForeignKey, DateTime,
Boolean, Text
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.sql import func

Base = declarative_base()

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    username = Column(String, unique=True, nullable=False)
    name = Column(String, nullable=False)
    email = Column(String, unique=True, nullable=False)
    password = Column(String, nullable=False)
    role = Column(String, nullable=False)
    created_at = Column(DateTime, server_default=func.now())

class MCPSession(Base):
```

```
    __tablename__ = "mcp_sessions"

    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey("users.id"), nullable=False)
    session_id = Column(String, unique=True, nullable=False)
    created_at = Column(DateTime, server_default=func.now())
    last_active_at = Column(DateTime, server_default=func.now(),
onupdate=func.now())
```

3. **Database Initialization**:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "sqlite:///./data/app.db"

engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

4. **Migration Script**:
    - Create a migration script to ensure compatibility
    - Consider using Alembic for future schema migrations

Considerations

- Ensure all SQLite-specific features have equivalent implementation
- Maintain data integrity during the migration
- Create database backup prior to migration

2. Authentication System Migration

Current Authentication

The application uses express-session for session management and bcrypt for password hashing.

Migration Steps

1. **JWT-Based Authentication**:

```
from datetime import datetime, timedelta
from jose import JWTError, jwt
from passlib.context import CryptContext
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer

SECRET_KEY = "your-secret-key"  # Load from config
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 1440  # 24 hours

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="auth/login")

def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password):
    return pwd_context.hash(password)

def create_access_token(data: dict):
    to_encode = data.copy()
    expire = datetime.utcnow() +
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

async def get_current_user(token: str = Depends(oauth2_scheme), db =
Depends(get_db)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        user_id: int = payload.get("sub")
        if user_id is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception

    user = db.query(User).filter(User.id == user_id).first()
    if user is None:
        raise credentials_exception
    return user
```

2. **Authentication Endpoints**:

```
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm
```

```
router = APIRouter(prefix="/auth", tags=["authentication"])

@router.post("/login")
async def login(form_data: OAuth2PasswordRequestForm = Depends(), db =
Depends(get_db)):
    user = db.query(User).filter(User.username ==
form_data.username).first()
    if not user or not verify_password(form_data.password, user.password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    access_token = create_access_token(data={"sub": str(user.id)})
    return {
        "access_token": access_token,
        "token_type": "bearer",
        "id": user.id,
        "username": user.username,
        "role": user.role
    }

@router.get("/me")
async def get_current_user_info(current_user = Depends(get_current_user)):
    return {
        "id": current_user.id,
        "username": current_user.username,
        "name": current_user.name,
        "email": current_user.email,
        "role": current_user.role
    }

@router.post("/logout")
async def logout():
    # With JWT, logout is handled client-side
    return {"message": "Logged out successfully"}
```

3. **Authentication Middleware**:

```
from fastapi import Security, HTTPException

async def admin_required(current_user = Security(get_current_user)):
    if current_user.role != "admin":
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Insufficient permissions"
        )
    return current_user
```

Considerations

- JWT tokens are stateless, unlike sessions; adjust the frontend accordingly
- Implement token refresh mechanism for long-lived sessions
- Consider Redis for token blacklisting if needed

3. API Endpoint Migration

Migration Process

1. **Endpoint Mapping**:
    - Document all existing Express.js routes
    - Map HTTP methods, URL paths, query parameters, and request bodies
    - Document expected responses and status codes
2. **FastAPI Route Implementation**:

```
from fastapi import APIRouter, Depends, HTTPException
from pydantic import BaseModel
from typing import List, Optional

# User model for request/response
class UserBase(BaseModel):
    username: str
    name: str
    email: str

class UserCreate(UserBase):
    password: str
    role: str = "user"

class UserResponse(UserBase):
    id: int
    role: str

    class Config:
        orm_mode = True

# Routes
router = APIRouter(prefix="/users", tags=["users"])

@router.get("/", response_model=List[UserResponse])
async def get_all_users(
    current_user = Depends(admin_required),
    db = Depends(get_db)
):
    users = db.query(User).all()
    return users

@router.post("/", response_model=UserResponse, status_code=201)
async def create_user(
```

```
    user: UserCreate,
    current_user = Depends(admin_required),
    db = Depends(get_db)
):
    # Check if username exists
    if db.query(User).filter(User.username == user.username).first():
        raise HTTPException(status_code=400, detail="Username already
exists")

    # Check if email exists
    if db.query(User).filter(User.email == user.email).first():
        raise HTTPException(status_code=400, detail="Email already exists")

    # Hash password
    hashed_password = get_password_hash(user.password)

    # Create user
    db_user = User(
        username=user.username,
        name=user.name,
        email=user.email,
        password=hashed_password,
        role=user.role
    )

    db.add(db_user)
    db.commit()
    db.refresh(db_user)

    return db_user
```

3. **Request/Response Validation with Pydantic**:
    - Define Pydantic models for all request and response types
    - Add validation and documentation
4. **Error Handling**:

```
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

app = FastAPI()

@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    return JSONResponse(
        status_code=500,
        content={"error": "Internal Server Error"}
    )
```

Considerations

- Leverage FastAPI's dependency injection for middleware functions
- Use Pydantic for request validation
- Match response formats exactly for backward compatibility with frontend

4. MCP Integration in Python

Current MCP Implementation

The current MCP implementation is planned using TypeScript SDK but will be migrated to Python.

Python MCP Implementation

1. **MCP Server Setup**:

```python
from modelcontextprotocol.server import MCPServer
from modelcontextprotocol.tools import Tool, ToolParameter
import subprocess
import shlex
import asyncio

# Define terminal command tool
terminal_command_tool = Tool(
    name="terminal_command",
    description="Execute a terminal command",
    parameters=[
        ToolParameter(
            name="command",
            description="The command to execute",
            type="string",
            required=True
        )
    ]
)

# Implement handler
async def handle_terminal_command(params, auth_context=None):
    command = params["command"]

    # Security validation (example)
    user_id = auth_context.get('user_id') if auth_context else None

    if not user_id:
        return {"error": "Authentication required"}

    # More security checks here...

    try:
        # Execute command safely
        args = shlex.split(command)
```

```
        process = await asyncio.create_subprocess_exec(
            *args,
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.PIPE,
            timeout=30
        )
        stdout, stderr = await process.communicate()

        return {
            "stdout": stdout.decode(),
            "stderr": stderr.decode(),
            "exit_code": process.returncode
        }
    except Exception as e:
        return {"error": str(e)}

# Create MCP server
server = MCPServer()
server.register_tool(terminal_command_tool, handle_terminal_command)
```

2. **MCP Client Integration**:

```
from modelcontextprotocol.client import MCPClient
from fastapi import APIRouter, Depends, HTTPException, WebSocket

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

# Initialize MCP client
async def get_mcp_client():
    client = MCPClient(transport_config={
        "type": "http",
        "url": "http://localhost:5000"
    })

    await client.connect()
    return client

# Execute MCP tool endpoint
@router.post("/execute")
async def execute_mcp_tool(
    tool_name: str,
    params: dict,
    current_user = Depends(get_current_user),
    mcp_client = Depends(get_mcp_client),
    db = Depends(get_db)
):
    # Security validation based on tool and user permissions
    # ...

    try:
        # Add tracking record
        cmd_record = MCPCommand(
```

```
            user_id=current_user.id,
            command=params.get("command", ""),
            status="executing"
        )
        db.add(cmd_record)
        db.commit()

        # Execute tool via MCP
        result = await mcp_client.execute_tool(
            tool_name,
            params,
            auth_context={"user_id": current_user.id}
        )

        # Update record
        cmd_record.result = result
        cmd_record.status = "executed" if "error" not in result else
"failed"
        db.commit()

        return result
    except Exception as e:
        # Update record with error
        if cmd_record:
            cmd_record.status = "failed"
            cmd_record.result = {"error": str(e)}
            db.commit()

        raise HTTPException(status_code=500, detail=str(e))
```

3. **WebSocket Integration for Real-time Updates**:

```
@router.websocket("/ws")
async def websocket_endpoint(
    websocket: WebSocket,
    token: str = None
):
    # Authenticate user from token
    try:
        if not token:
            await websocket.close(code=1008)
            return

        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        user_id = payload.get("sub")

        if not user_id:
            await websocket.close(code=1008)
            return
    except:
        await websocket.close(code=1008)
        return
```

```
    await websocket.accept()

    # Handle WebSocket connection
    try:
        while True:
            data = await websocket.receive_json()

            # Process command
            if data.get("type") == "execute_command":
                # Initialize MCP client
                mcp_client = await get_mcp_client()

                # Execute command
                result = await mcp_client.execute_tool(
                    data["tool"],
                    data["params"],
                    auth_context={"user_id": user_id}
                )

                # Send result back through WebSocket
                await websocket.send_json({
                    "type": "command_result",
                    "id": data.get("id"),
                    "result": result
                })
    except Exception as e:
        # Handle WebSocket errors
        await websocket.close(code=1011)
```

MCP Integration Benefits in Python

1. **Native Async Support**: Python's asyncio works seamlessly with FastAPI and MCP
2. **Official SDK**: The Python SDK for MCP is the most mature and feature-complete
3. **Process Management**: Python's subprocess module gives better control for terminal commands
4. **Security**: Python has robust libraries for sandboxing and secure command execution
5. **AI Ecosystem**: Integration with other AI libraries (if needed) is smoother in Python

5. Configuration and Deployment

Configuration Management

Migrate from config.ini to environment variables and Python's configuration management:

```
from pydantic import BaseSettings
from functools import lru_cache
import os

class Settings(BaseSettings):
    # Server settings
```

```
    server_host: str = "0.0.0.0"
    server_port: int = 5634

    # Database settings
    database_url: str = "sqlite:///./data/app.db"

    # Security settings
    secret_key: str
    token_expire_minutes: int = 1440

    # Admin settings
    default_admin_username: str = "admin"
    default_admin_password: str = "admin"
    default_admin_email: str = "admin@localhost"

    # MCP settings
    mcp_server_url: str = "http://localhost:5000"

    class Config:
        env_file = ".env"

@lru_cache()
def get_settings():
    return Settings()
```

Deployment Strategy

1. **Docker Containerization**:

```
# Dockerfile
FROM python:3.10

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "5634"]
```

2. **Docker Compose for Local Development**:

```
version: '3'

services:
  backend:
    build: .
    ports:
```

```
      - "5634:5634"
    volumes:
      - ./data:/app/data
    environment:
      - SECRET_KEY=your_development_secret
      - DATABASE_URL=sqlite:///./data/app.db
    command: uvicorn main:app --reload --host 0.0.0.0 --port 5634

  mcp-server:
    build: ./mcp-server
    ports:
      - "5000:5000"
    volumes:
      - ./data:/app/data
```

6. Migration Timeline and Phases

Phase 1: Preparation and Setup (1-2 weeks)

1. Set up Python development environment
2. Create FastAPI project structure
3. Implement database models with SQLAlchemy
4. Document all existing endpoints for migration

Phase 2: Core Components (2-3 weeks)

1. Implement authentication system
2. Migrate basic CRUD endpoints
3. Set up error handling and middleware
4. Create database migration scripts

Phase 3: MCP Integration (2-3 weeks)

1. Implement MCP server in Python
2. Create MCP client integration with FastAPI
3. Implement WebSocket for real-time updates
4. Build command execution and approval flows

Phase 4: Testing and Optimization (1-2 weeks)

1. Write comprehensive tests for all endpoints
2. Perform load testing and optimize performance
3. Implement monitoring and logging
4. Address any security concerns

Phase 5: Deployment and Transition (1-2 weeks)

1. Set up CI/CD pipeline for Python backend
2. Create deployment scripts

3. Transition traffic gradually from Node.js to Python
4. Monitor for any issues and fix as needed

Pros and Cons of Migration

Pros

1. **Better MCP Integration**: Native Python SDK for MCP provides more reliable integration
2. **Stronger Type System**: With Pydantic and type hints, Python can provide robust validation
3. **Async Performance**: FastAPI's async capabilities handle concurrent connections efficiently
4. **Auto-Documentation**: FastAPI generates OpenAPI documentation automatically
5. **AI Ecosystem**: Python has better libraries for AI/ML if needed in future
6. **Terminal Command Handling**: Python offers more secure and flexible subprocess management

Cons

1. **Migration Effort**: Substantial effort required to rewrite backend components
2. **Potential Bugs**: Risk of introducing new bugs during migration
3. **Team Adaptation**: Team may need to adapt to Python ecosystem if primarily Node.js focused
4. **Temporary System Complexity**: Running both systems in parallel adds complexity
5. **Frontend Integration**: May require updates to frontend API calls if response formats change

Risk Mitigation

1. **Comprehensive Testing**: Implement thorough testing of all migrated components
2. **Phased Migration**: Migrate one component at a time to minimize risk
3. **Feature Parity**: Ensure all existing functionality is replicated before cutover
4. **Rollback Plan**: Maintain ability to revert to Node.js backend if issues arise
5. **Documentation**: Create detailed documentation of the new Python backend

Conclusion

Migrating from Node.js to Python offers significant advantages for MCP integration in the co-pilot application. While the migration requires substantial effort, the benefits in terms of maintainability, performance, and integration capabilities make it worthwhile.

By following this phased approach and leveraging modern Python frameworks like FastAPI, the migration can be accomplished with minimal disruption to users and optimal long-term results. The Python ecosystem's strengths in AI, process management, and strong typing align perfectly with the requirements of an MCP-based co-pilot application.

This migration not only addresses immediate needs for MCP integration but also positions the application for future enhancements and integrations with the broader AI ecosystem.