PM Project Tracker - Complete Developer Guide

PyWebView Native Desktop Edition

Version - Enterprise Integration & Native Desktop
Author: William Nash (william.nash@verizon.com)
Organization: Verizon Internal Systems

Table of Contents

---

1. Executive Summary

1.1 Project Overview

The PM Project Tracker is an enterprise-grade **native desktop application** designed to be a comprehensive toolkit for project management at Verizon. Built with PyWebView, it provides a modern, user-centric interface that looks and feels like a true desktop application while leveraging web technologies for the UI. It connects directly to the Golden Source (GS) Oracle Database, delivering unprecedented efficiency gains while maintaining enterprise security standards.

1.2 Key Objectives

- **Centralize Project Management**: Single pane of glass for all PM activities

- **Native Desktop Experience**: True desktop application, not browser-based
- **No External Dependencies**: Single executable with everything bundled
- **Seamless Integration**: One-click integrations with Slack, Webex, Google Workspace
- **Foster Team Collaboration**: Task assignment, commenting, and shared visibility
- **Modern User Experience**: Intuitive, customizable interface
- **Empower with Data**: Custom reporting engine and visual Gantt charts
- **Voice Capabilities**: Text-to-speech for accessibility and hands-free updates

1.3 Why PyWebView?

PyWebView was chosen because it:

- Creates a native desktop window (not a browser tab)
- Requires no external browser installation
- Bundles into a single executable
- Provides full desktop integration (file dialogs, notifications, system tray)
- Works within Verizon's constraints (no API keys, no external software)
- Maintains modern web UI capabilities

---

2. System Architecture & Design

2.1 Native Desktop Architecture

The application uses a **simplified two-tier architecture** powered by PyWebView:

```
┌─────────────────────────────────────────────┐
│        PMTracker.exe (Single Process)        │
│                                              │
│  ┌───────────────────────────────────────┐  │
│  │    PyWebView Native Window             │  │
│  │  (System WebView - EdgeHTML/WebKit)    │  │
│  │                                        │  │
│  │  ┌─────────────────────────────────┐  │  │
│  │  │  Modern Web UI                   │  │  │
│  │  │  - HTML5/CSS3/JavaScript         │  │  │
│  │  │  - Tailwind CSS                  │  │  │
│  │  │  - GridStack.js Dashboard        │  │  │
│  │  │  - DHTMLX Gantt Charts           │  │  │
│  │  └─────────────────────────────────┘  │  │
│  │                                        │  │
│  │        ↕ JavaScript API Bridge         │  │
│  │                                        │  │
│  └───────────────────────────────────────┘  │
│              ↕                               │
│  ┌───────────────────────────────────────┐  │
│ │  FastAPI Backend (Background Thread)   │ │
│ │ - Business Logic                       │ │
│ │ - Data Processing                      │ │
│ │ - Database Connections                 │ │
│ │ - ML Model Inference                   │ │
│ │ - TTS Engine                           │ │
│ └────────────────────────────────────────┘ │
│              ↕                              │
│ ┌────────────────────────────────────────┐ │
│ │       Data Layer                       │ │
│ │ - Oracle GS Database (Read-Only)       │ │
│ │ - SQLite (pm_tracker_user.db)          │ │
│ │ - G: Drive (ML Models, Updates)        │ │
│ └────────────────────────────────────────┘ │
└─────────────────────────────────────────────┘
```

Architecture Layers:

1. **Presentation Layer (PyWebView Window)**
   - Native desktop window with embedded web view
   - Responsive HTML/CSS/JS interface
   - Customizable widget-based dashboard
   - Interactive Gantt charts
   - Real-time updates via WebSocket
2. **Application Layer (Python Backend)**
   - FastAPI running in background thread
   - Business logic and data processing
   - Oracle and SQLite database connections
   - ML model serving
   - Text-to-speech engine
   - Auto-updater service
3. **Data Layer (Hybrid Storage)**
   - **Golden Source Oracle Database**: Primary data source (read-only)
   - **Local SQLite Database**: User-generated content (notes, tasks, settings)
   - **G: Drive**: Shared resources (ML models, updates, documentation)

2.2 User Experience (UX) and Interface (UI) Design

2.2.1 Customizable Dashboard

The main dashboard features a fully customizable, widget-based layout:

Core Widgets:

- KPI Cards (project counts, completion rates, delays)
- Project Lists (filterable by status, PM, team)
- Task Lists (personal and team tasks)
- Analytics Charts (timeline trends, risk distribution)

Utility Widgets:

- World Clock (multiple time zones)
- Local Weather (Azle, TX)
- Quick Links (internal sites)
- Personal Notes

Integration Widgets:

- TCOMS Quick View
- Slack Activity Feed
- Upcoming Meetings
- Recent Documents

2.2.2 In-App Help & Guidance

Contextual Help:

- Tooltip explanations on hover
- Help icons next to complex features
- Inline examples and suggestions

Guided Tours:

- First-time user walkthrough
- Feature discovery prompts
- Interactive tutorials (powered by Shepherd.js)

Central Help Menu:

- Searchable documentation
- Video tutorials (hosted on G: drive)
- FAQ section
- Contact support

2.2.3 Template Engine

Project Note Templates:

- Weekly status updates
- Project kickoff notes
- Closure documentation
- Risk assessments

Report Templates:

- Executive summary
- Detailed technical reports
- Performance dashboards
- Resource utilization

Workflow Templates:

- New project kickoff sequence
- Status update routine
- Escalation procedures
- Handoff checklists

2.3 Modular Project Structure

```
PMTracker/
│
├── src/
│   ├── main.py                  # PyWebView entry point
│   │
│   ├── api/                   # FastAPI routers
│   │   ├── __init__.py
│   │   ├── projects.py
│   │   ├── reports.py
│   │   ├── gantt.py
│   │   ├── tasks.py
│   │   ├── notes.py
│   │   ├── ml.py
│   │   └── tts.py
│   │
│   ├── core/                   # Core components
│   │   ├── __init__.py
│   │   ├── oracle_manager.py       # Oracle connection pool
│   │   ├── sqlite_manager.py       # SQLite operations
│   │   └── config_manager.py       # Configuration handling
│   │
│   ├── models/                  # Pydantic models
│   │   ├── __init__.py
│   │   ├── project.py
│   │   ├── task.py
│   │   ├── note.py
│   │   └── report.py
│   │
│   ├── services/                # Business logic
│   │   ├── __init__.py
│   │   ├── project_service.py
│   │   ├── task_service.py
│   │   ├── note_service.py
│   │   └── report_service.py
│   │
│   ├── integrations/          # External tool connectors
│   │   ├── __init__.py
│   │   ├── slack.py
│   │   ├── webex.py
│   │   ├── google_workspace.py
│   │   └── tcoms.py
│   │
│   ├── ml/                    # Machine learning
│   │   ├── __init__.py
│   │   ├── delay_predictor.py
│   │   └── risk_classifier.py
│   │
│   ├── reporting/             # Report generation
│   │   ├── __init__.py
│   │   ├── engine.py
│   │   ├── exporters.py
│   │   └── templates/
│   │
│   ├── gantt/                 # Gantt chart logic
│   │   ├── __init__.py
│   │   └── generator.py
│   │
│   ├── notes/                 # Project notebook
│   │   ├── __init__.py
│   │   └── manager.py
│   │
│   ├── tasks/                 # Task management
│   │   ├── __init__.py
│   │   └── manager.py
│   │
│   ├── collaboration/         # Comments & mentions
│   │   ├── __init__.py
│   │   └── comment_system.py
│   │
│   ├── communication/         # Email & notifications
│   │   ├── __init__.py
│   │   └── email_generator.py
│   │
│   ├── web_app/               # Frontend assets
│   │   ├── static/
│   │   │   ├── css/
│   │   │   │   ├── main.css
│   │   │   │   └── themes/
│   │   │   ├── js/
│   │   │   │   ├── dashboard.js
│   │   │   │   ├── gantt.js
│   │   │   │   ├── projects.js
│   │   │   │   └── utils.js
│   │   │   └── assets/
│   │   │       ├── images/
│   │   │       └── icons/
│   │   └── templates/
│   │       ├── index.html
│   │       ├── dashboard.html
│   │       ├── projects.html
│   │       ├── gantt.html
│   │       ├── reports.html
│   │       └── settings.html
│   │
│   └── utils/                 # Utilities
│       ├── __init__.py
│       ├── logger.py
│       ├── tts_manager.py      # Text-to-speech
│       ├── window_manager.py   # Window controls
│       ├── updater.py          # Auto-update logic
│       └── helpers.py
│
├── resources/                 # Application resources
│   ├── icon.ico
│   ├── splash.png
│   └── sounds/
│
├── config/                    # Configuration files
│   ├── config.ini.template
│   └── logging.yaml
│
├── tests/                     # Unit tests
│   ├── test_api.py
│   ├── test_services.py
│   └── test_integrations.py
│
├── scripts/                   # Utility scripts
│   ├── build.py
│   ├── update_version.py
│   └── setup_dev.py
│
├── requirements.txt           # Dependencies
├── build.spec                 # PyInstaller config
└── README.md
```

3. Technology Stack

3.1 Core Technologies

Desktop Framework:

- **PyWebView 4.4+**: Native window creation and web view embedding
- **PySide6**: Optional configuration dialogs

Backend:

- **Python 3.11+**: Application runtime
- **FastAPI 0.104+**: REST API framework
- **Uvicorn 0.24+**: ASGI server
- **Pydantic 2.5+**: Data validation

Database:

- **oracledb 2.0+**: Oracle database connector (thin mode)
- **sqlite3**: Built-in Python module for local storage

Frontend:

- **HTML5/CSS3/JavaScript (ES6+)**
- **Tailwind CSS 3.3+**: Utility-first CSS framework
- **Lucide Icons**: Icon library
- **GridStack.js 9.0+**: Drag-and-drop dashboard
- **DHTMLX Gantt 8.0+**: Interactive Gantt charts
- **Shepherd.js 11.0+**: Guided tours
- **Chart.js 4.0+**: Data visualization

Text-to-Speech:

- **gTTS 2.4+**: Google Text-to-Speech
- **pydub 0.25+**: Audio processing
- **pywin32**: Windows audio playback

Machine Learning:

- **TensorFlow/Keras 2.15+**: ML framework
- **Pandas 2.1+**: Data manipulation
- **Scikit-learn 1.3+**: ML utilities
- **NumPy 1.26+**: Numerical computing

Utilities:

- **cryptography 41.0+**: Credential encryption

- **python-dotenv 1.0+**: Environment management
- **pyperclip 1.8+**: Clipboard operations
- **Jinja2 3.1+**: Template engine
- **openpyxl 3.1+**: Excel file generation
- **reportlab 4.0+**: PDF generation

Build & Deployment:

- **PyInstaller 6.0+**: Executable creation
- **win10toast**: Windows notifications

3.2 Complete Requirements

```
# requirements.txt

# Core Framework
pywebview==4.4.1
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
pydantic-settings==2.1.0

# Database
oracledb==2.0.0
# sqlite3 is built-in

# Frontend Assets (served locally)
jinja2==3.1.2

# Text-to-Speech
gtts==2.4.0
pydub==0.25.1
pywin32==306

# Machine Learning
tensorflow==2.15.0
keras==2.15.0
pandas==2.1.4
scikit-learn==1.3.2
numpy==1.26.2

# Security
cryptography==41.0.7
python-dotenv==1.0.0

# Utilities
pyperclip==1.8.2
openpyxl==3.1.2
```

```
reportlab==4.0.7
pillow==10.1.0
requests==2.31.0
python-multipart==0.0.6

# Windows Integration
win10toast==0.9

# Development
pytest==7.4.3
pytest-asyncio==0.21.1
black==23.12.0
flake8==6.1.0
```

---

4. Oracle Database Integration

4.1 Golden Source (GS) Schema Overview

The application connects to the **SPLUNKVEEP_NAR** Oracle user, which hosts the Golden Source (GS) database schema. This schema consolidates data from various Verizon systems into a structured, reliable format organized into logical "layers" by business domain.

Connection Details:

- **Host**: f1btpap-scan.verizon.com
- **Service**: NARPROD
- **User**: splunkveep_nar
- **Mode**: Thin (no Oracle Client required)

4.2 Key Data Layers & Tables

Layer	Primary Tables	Description
WFM	`GS_WFM_NF_PROJECTS`	Core Canvas WFM projects (NFIDs, status, type)
	`GS_WFM_NF_PRJ_MILESTONES`	Key project milestones
	`GS_WFM_UTE_TASKS`	UTE task details
	`GSC_WFM_NFID_DEPENDENCY_SUMMARY`	Project dependencies
CCP	`GS_CCP_CCRS`	Customer Carrier Requests (CCRs)

	`GS_CCP_CCR_QUEUES`	VZB Local CCR queues
	`GS_CCP_TEOS`	Telecommunications Equipment Orders
	`GS_CCP_CCR_KEY_MILESTONE_DATES`	Milestone tracking
PMRA	`GS_PMRA_PROV_ORDERS`	Provisioning order data
DECOM	`GS_DECOM_*`	Decommissioning assets and sites
REF	`GS_REF_HR_DETAILS`	Employee information & hierarchy
	`GS_REF_WORKDAY_CALENDAR`	Business day calculations
AYS	`GS_AYS_TICKETS`	AYS ticketing system data
PORCH	`GS_PORCH_CUSTOMER_ORDERS`	Customer order details

4.3 ETL Architecture and Process

4.3.1 ETL Framework

Data pipelines are built using standardized Python framework:

- **Location**: `gsetl_project/etl/`
- **Scripts**: `Refresher_*.py` (one per data layer)
- **Process**: Extract → Transform (Pandas) → Load (SQL MERGE)

4.3.2 Job Orchestration with Airflow

- **Tool**: Apache Airflow
- **DAGs**: `gsetl_project/customized_dags/`
- **Purpose**: Schedule and manage dependencies

4.3.3 Data Refresh Cadence

Layer	Frequency	Timing
CCP, WFM, PMRA, PORCH	Daily	Early morning

REF (HR Data)	Daily	Early morning
DECOM	Weekly	Monday
AYS	Daily	Early morning

4.4 Connection Architecture

Implementation (`src/core/oracle_manager.py`):

```
import oracledb
from contextlib import contextmanager

class OracleManager:
    def __init__(self, config):
        self.config = config
        self.pool = None
        self._initialize_pool()

    def _initialize_pool(self):
        """Create connection pool"""
        try:
            self.pool = oracledb.create_pool(
                user=self.config.oracle_user,
                password=self.config.oracle_password,
                dsn=f"{self.config.oracle_host}/{self.config.oracle_service}",
                min=2,
                max=10,
                increment=1,
                threaded=True
            )
        except Exception as e:
            raise ConnectionError(f"Failed to create Oracle pool: {e}")

    @contextmanager
    def get_connection(self):
        """Get connection from pool"""
        conn = self.pool.acquire()
        try:
            yield conn
        finally:
            self.pool.release(conn)

    def execute_query(self, query, params=None):
        """Execute SELECT query"""
        with self.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(query, params or {})
            columns = [col[0] for col in cursor.description]
            results = cursor.fetchall()
            return [dict(zip(columns, row)) for row in results]

    def test_connection(self):
        """Test database connectivity"""
        try:
            with self.get_connection() as conn:
                cursor = conn.cursor()
                cursor.execute("SELECT 1 FROM DUAL")
                cursor.fetchone()
            return True
        except Exception as e:
            return False
```

4.5 Credential Management

Encryption Implementation (`src/core/config_manager.py`):

```
from cryptography.fernet import Fernet
import os
import configparser

class ConfigManager:
    def __init__(self, config_path='config/config.ini'):
        self.config_path = config_path
        self.key_path = os.path.join(os.path.expanduser('~'), '.pmtracker_key')
        self.cipher = self._get_cipher()
        self.config = self._load_config()

    def _get_cipher(self):
        """Get or create encryption key"""
        if os.path.exists(self.key_path):
            with open(self.key_path, 'rb') as f:
                key = f.read()
        else:
            key = Fernet.generate_key()
            with open(self.key_path, 'wb') as f:
                f.write(key)
        return Fernet(key)

    def encrypt_password(self, password):
        """Encrypt password"""
        return self.cipher.encrypt(password.encode()).decode()

    def decrypt_password(self, encrypted_password):
        """Decrypt password"""
        return self.cipher.decrypt(encrypted_password.encode()).decode()

    def save_credentials(self, username, password):
        """Save encrypted credentials"""
        config = configparser.ConfigParser()
        config['Oracle'] = {
            'username': username,
            'password': self.encrypt_password(password),
            'host': 'f1btpap-scan.verizon.com',
            'service': 'NARPROD'
        }
        with open(self.config_path, 'w') as f:
            config.write(f)

    def get_credentials(self):
        """Get decrypted credentials"""
        config = configparser.ConfigParser()
        config.read(self.config_path)
        return {
            'username': config['Oracle']['username'],
            'password': self.decrypt_password(config['Oracle']['password']),
            'host': config['Oracle']['host'],
            'service': config['Oracle']['service']
        }
```

---

5. Application Components

5.1 Main Application Entry Point

`src/main.py` - PyWebView initialization:

```
import webview
import threading
import sys
import os
from pathlib import Path

# Add src to path
sys.path.insert(0, str(Path(__file__).parent))

from api import create_app
from utils.window_manager import WindowManager
from utils.updater import check_for_updates
from core.config_manager import ConfigManager
import uvicorn

class PMTrackerApp:
    def __init__(self):
        self.window = None
        self.api = None
        self.config = ConfigManager()

    def start_backend(self):
        """Start FastAPI in background thread"""
        app = create_app()
        uvicorn.run(
            app,
            host="127.0.0.1",
            port=8000,
            log_level="error",
            access_log=False
        )

    def on_closing(self):
        """Handle window close event"""
        # Save window state
        # Cleanup resources
        return True

    def run(self):
        """Main application entry point"""
        # Check for updates
        update_available = check_for_updates()
        if update_available:
            # Handle update
            pass

        # Start FastAPI in background thread
        backend_thread = threading.Thread(target=self.start_backend, daemon=True)
        backend_thread.start()

        # Create JavaScript API
        self.api = WindowManager()

        # Create PyWebView window
        self.window = webview.create_window(
            'PM Project Tracker',
            'http://127.0.0.1:8000',
            width=1400,
            height=900,
            resizable=True,
            fullscreen=False,
            min_size=(1024, 768),
            confirm_close=True,
            on_top=False,
            js_api=self.api
        )

        # Start PyWebView
        webview.start(
            debug=False,
            http_server=False
        )

if __name__ == '__main__':
    app = PMTrackerApp()
    app.run()
```

5.2 FastAPI Application Factory

`src/api/__init__.py`:

```
from fastapi import FastAPI
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from fastapi.middleware.cors import CORSMiddleware

from .projects import router as projects_router
from .reports import router as reports_router
from .gantt import router as gantt_router
from .tasks import router as tasks_router
from .notes import router as notes_router
from .ml import router as ml_router
from .tts import router as tts_router

def create_app():
    app = FastAPI(
        title="PM Project Tracker API",
        version="10.1.0"
    )

    # CORS for local access
    app.add_middleware(
        CORSMiddleware,
        allow_origins=["http://127.0.0.1:8000"],
        allow_credentials=True,
```

```python
        allow_methods=["*"],
        allow_headers=["*"],
    )

    # Mount static files
    app.mount("/static", StaticFiles(directory="web_app/static"), name="static")

    # Include routers
    app.include_router(projects_router, prefix="/api/projects", tags=["projects"])
    app.include_router(reports_router, prefix="/api/reports", tags=["reports"])
    app.include_router(gantt_router, prefix="/api/gantt", tags=["gantt"])
    app.include_router(tasks_router, prefix="/api/tasks", tags=["tasks"])
    app.include_router(notes_router, prefix="/api/notes", tags=["notes"])
    app.include_router(ml_router, prefix="/api/ml", tags=["ml"])
    app.include_router(tts_router, prefix="/api/tts", tags=["tts"])

    # Health check
    @app.get("/api/health")
    async def health_check():
        return {"status": "healthy", "version": "10.1.0"}

    return app
```

5.3 Window Manager with JavaScript Bridge

`src/utils/window_manager.py`:

```python
import webview
import pyperclip
import webbrowser
from win10toast import ToastNotifier

class WindowManager:
    """JavaScript API exposed to frontend"""

    def __init__(self):
        self.toaster = ToastNotifier()

    def minimize_window(self):
        """Minimize application window"""
        if webview.windows:
            webview.windows[0].minimize()

    def maximize_window(self):
        """Toggle fullscreen"""
        if webview.windows:
            webview.windows[0].toggle_fullscreen()
    def close_window(self):
        """Close application"""
        if webview.windows:
            webview.windows[0].destroy()

    def open_file_dialog(self, file_types=None):
        """Open native file dialog"""
        if not webview.windows:
            return None

        result = webview.windows[0].create_file_dialog(
            webview.OPEN_DIALOG,
            allow_multiple=False,
            file_types=file_types or ('All Files (*.*)',)
        )
        return result[0] if result else None

    def save_file_dialog(self, filename='', file_types=None):
        """Open save file dialog"""
        if not webview.windows:
            return None

        result = webview.windows[0].create_file_dialog(
            webview.SAVE_DIALOG,
            save_filename=filename,
            file_types=file_types or ('All Files (*.*)',)
        )
        return result if result else None

    def show_notification(self, title, message, duration=5):
        """Show Windows toast notification"""
        try:
            self.toaster.show_toast(
                title,
                message,
                duration=duration,
                threaded=True
            )
            return {"status": "success"}
        except Exception as e:
            return {"status": "error", "message": str(e)}

    def copy_to_clipboard(self, text):
        """Copy text to clipboard"""
        try:
            pyperclip.copy(text)
            return {"status": "success"}
        except Exception as e:
            return {"status": "error", "message": str(e)}

    def open_url(self, url):
        """Open URL in default browser"""
        try:
            webbrowser.open(url)
            return {"status": "success"}
        except Exception as e:
            return {"status": "error", "message": str(e)}

    def get_app_version(self):
        """Get application version"""
        return {"version": "10.1.0"}
```

5.4 Text-to-Speech Manager

`src/utils/tts_manager.py`:

```
from gtts import gTTS
import tempfile
import os
import threading
from pydub import AudioSegment
from pydub.playback import play

class TTSManager:
    """Text-to-Speech manager using Google TTS"""

    def __init__(self):
        self.is_speaking = False
        self.current_thread = None

    def speak(self, text, language='en', slow=False):
        """Convert text to speech and play it"""
        if self.is_speaking:
            return {"status": "busy", "message": "Already speaking"}

        self.current_thread = threading.Thread(
            target=self._speak_thread,
            args=(text, language, slow)
        )
        self.current_thread.daemon = True
        self.current_thread.start()

        return {"status": "speaking"}

    def _speak_thread(self, text, language, slow):
        """Background thread for TTS"""
        self.is_speaking = True
        temp_file = None

        try:
            # Generate speech
            tts = gTTS(text=text, lang=language, slow=slow)

            # Save to temporary file
            with tempfile.NamedTemporaryFile(delete=False, suffix='.mp3') as fp:
                temp_file = fp.name
                tts.save(temp_file)

            # Play audio
            audio = AudioSegment.from_mp3(temp_file)
            play(audio)

        except Exception as e:
            print(f"TTS Error: {e}")
        finally:
            # Cleanup
            if temp_file and os.path.exists(temp_file):
                try:
                    os.unlink(temp_file)
                except:
                    pass
            self.is_speaking = False

    def stop(self):
        """Stop current speech"""
        if self.current_thread and self.current_thread.is_alive():
            # Note: Can't easily stop pydub playback
            # Would need more complex implementation
            pass
        self.is_speaking = False
        return {"status": "stopped"}

    def get_status(self):
        """Get current TTS status"""
        return {"is_speaking": self.is_speaking}

# Global TTS instance
tts_manager = TTSManager()
```

`src/api/tts.py`:

```
from fastapi import APIRouter, HTTPException
from pydantic import BaseModel
from utils.tts_manager import tts_manager

router = APIRouter()

class TTSRequest(BaseModel):
    text: str
    language: str = 'en'
    slow: bool = False

@router.post("/speak")
async def speak(request: TTSRequest):
    """Text-to-speech endpoint"""
    result = tts_manager.speak(request.text, request.language, request.slow)
    return result

@router.post("/stop")
async def stop_speaking():
    """Stop current speech"""
    result = tts_manager.stop()
    return result

@router.get("/status")
async def get_status():
    """Get TTS status"""
    return tts_manager.get_status()
```

---

6. Core Features & PM Toolkit

6.1 Advanced Reporting & Analytics

Custom Report Builder (`src/reporting/engine.py`):

```
import pandas as pd
from datetime import datetime, timedelta
from openpyxl import Workbook
from openpyxl.styles import Font, PatternFill, Alignment
from reportlab.lib.pagesizes import letter
from reportlab.platypus import SimpleDocTemplate, Table, TableStyle, Paragraph
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.lib import colors

class ReportEngine:
    def __init__(self, oracle_manager):
        self.db = oracle_manager

    def build_custom_report(self, config):
        """Build custom report based on configuration"""
        # Extract config
        data_source = config.get('data_source')  # 'wfm', 'ccp', 'pmra'
        columns = config.get('columns', [])
        filters = config.get('filters', {})
        sort_by = config.get('sort_by', [])
        timeframe = config.get('timeframe', 'monthly')

        # Build query
        query = self._build_query(data_source, columns, filters, timeframe)

        # Execute query
        data = self.db.execute_query(query)

        # Convert to DataFrame
        df = pd.DataFrame(data)

        # Apply sorting
        if sort_by:
            df = df.sort_values(by=sort_by)

        return df

    def _build_query(self, data_source, columns, filters, timeframe):
        """Build SQL query based on parameters"""
        # Map data sources to tables
        table_map = {
            'wfm': 'GS_WFM_NF_PROJECTS',
            'ccp': 'GS_CCP_CCRS',
            'pmra': 'GS_PMRA_PROV_ORDERS'
        }

        table = table_map.get(data_source)

        # Build column list
        column_str = ', '.join(columns) if columns else '*'

        # Build WHERE clause
        where_clauses = []

        # Timeframe filter
        if timeframe == 'weekly':
            where_clauses.append(f"CREATE_DATE >= SYSDATE - 7")
        elif timeframe == 'monthly':
            where_clauses.append(f"CREATE_DATE >= SYSDATE - 30")
        elif timeframe == 'quarterly':
            where_clauses.append(f"CREATE_DATE >= SYSDATE - 90")
        elif timeframe == 'yearly':
            where_clauses.append(f"CREATE_DATE >= SYSDATE - 365")

        # Custom filters
        for key, value in filters.items():
            if isinstance(value, list):
                values = "', '".join(str(v) for v in value)
                where_clauses.append(f"{key} IN ('{values}')")
            else:
                where_clauses.append(f"{key} = '{value}'")

        where_str = ' AND '.join(where_clauses) if where_clauses else '1=1'

        query = f"SELECT {column_str} FROM {table} WHERE {where_str}"

        return query

    def export_to_excel(self, df, filename):
        """Export DataFrame to Excel with formatting"""
        wb = Workbook()
        ws = wb.active
        ws.title = "Report"

        # Header styling
        header_fill = PatternFill(start_color="366092", end_color="366092", fill_type="solid")
        header_font = Font(color="FFFFFF", bold=True)

        # Write headers
        for col_idx, column in enumerate(df.columns, 1):
            cell = ws.cell(row=1, column=col_idx, value=column)
            cell.fill = header_fill
            cell.font = header_font
            cell.alignment = Alignment(horizontal="center")

        # Write data
        for row_idx, row in enumerate(df.itertuples(index=False), 2):
            for col_idx, value in enumerate(row, 1):
                ws.cell(row=row_idx, column=col_idx, value=value)

        # Auto-adjust column widths
        for column in ws.columns:
            max_length = 0
            column_letter = column[0].column_letter
            for cell in column:
                try:
                    if len(str(cell.value)) > max_length:
                        max_length = len(cell.value)
                except:
                    pass
            adjusted_width = min(max_length + 2, 50)
            ws.column_dimensions[column_letter].width = adjusted_width

        wb.save(filename)
        return filename

    def export_to_pdf(self, df, filename, title="Project Report"):
        """Export DataFrame to PDF"""
        doc = SimpleDocTemplate(filename, pagesize=letter)
        elements = []
        styles = getSampleStyleSheet()

        # Title
        title_para = Paragraph(title, styles['Title'])
        elements.append(title_para)

        # Convert DataFrame to list for table
        data = [df.columns.tolist()] + df.values.tolist()

        # Create table
        table = Table(data)
        table.setStyle(TableStyle([
            ('BACKGROUND', (0, 0), (-1, 0), colors.grey),
            ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
            ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
            ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
            ('FONTSIZE', (0, 0), (-1, 0), 10),
            ('BOTTOMPADDING', (0, 0), (-1, 0), 12),
            ('BACKGROUND', (0, 1), (-1, -1), colors.beige),
            ('GRID', (0, 0), (-1, -1), 1, colors.black)
        ]))

        elements.append(table)
        doc.build(elements)
        return filename

    def generate_performance_report(self, timeframe='monthly'):
        """Generate performance summary report"""
        query = f"""
        SELECT
            STATUS,
            COUNT(*) as PROJECT_COUNT,
            AVG(DAYS_OPEN) as AVG_DAYS_OPEN,
            SUM(CASE WHEN DELAY_FLAG = 'Y' THEN 1 ELSE 0 END) as
```

```
DELAYED_COUNT
        FROM GS_WFM_NF_PROJECTS
        WHERE CREATE_DATE >= SYSDATE - {self._get_days(timeframe)}
        GROUP BY STATUS
        ORDER BY PROJECT_COUNT DESC
        """

        data = self.db.execute_query(query)
        return pd.DataFrame(data)

    def _get_days(self, timeframe):
        """Convert timeframe to days"""
        timeframe_map = {
            'weekly': 7,
            'monthly': 30,
            'quarterly': 90,
            'yearly': 365
        }
        return timeframe_map.get(timeframe, 30)
```

API Endpoint (`src/api/reports.py`):

```
from fastapi import APIRouter, HTTPException
from pydantic import BaseModel
from typing import List, Dict, Any, Optional
from services.report_service import ReportService
from core.oracle_manager import OracleManager
from core.config_manager import ConfigManager

router = APIRouter()

# Initialize services
config = ConfigManager()
oracle = OracleManager(config.get_credentials())
report_service = ReportService(oracle)

class ReportConfig(BaseModel):
    data_source: str
    columns: List[str]
    filters: Dict[str, Any] = {}
    sort_by: List[str] = []
    timeframe: str = 'monthly'

class ExportRequest(BaseModel):
    report_id: str
    format: str  # 'excel' or 'pdf'
    filename: Optional[str] = None
```

```

@router.post("/build")
async def build_report(config: ReportConfig):
    """Build custom report"""
    try:
        df = report_service.build_custom_report(config.dict())
        return {
            "status": "success",
            "data": df.to_dict(orient='records'),
            "row_count": len(df)
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@router.post("/export")
async def export_report(request: ExportRequest):
    """Export report to file"""
    try:
        result = report_service.export_report(
            request.report_id,
            request.format,
            request.filename
        )
        return result
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@router.get("/performance")
async def get_performance_report(timeframe: str = 'monthly'):
    """Get performance summary report"""
    try:
        df = report_service.generate_performance_report(timeframe)
        return {
            "status": "success",
            "data": df.to_dict(orient='records')
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

6.2 Integrated Project Notebook

Note Manager (`src/notes/manager.py`):

```
import sqlite3
from datetime import datetime
from typing import List, Dict, Optional

```

```
class NoteManager:
    def __init__(self, db_path='pm_tracker_user.db'):
        self.db_path = db_path
        self._initialize_db()

    def _initialize_db(self):
        """Create notes table if not exists"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS notes (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                project_id VARCHAR(50) NOT NULL,
                title VARCHAR(255) NOT NULL,
                content TEXT NOT NULL,
                tags VARCHAR(255),
                created_by VARCHAR(100),
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
        """)
        cursor.execute("""
            CREATE INDEX IF NOT EXISTS idx_notes_project
            ON notes(project_id)
        """)
        cursor.execute("""
            CREATE INDEX IF NOT EXISTS idx_notes_tags
            ON notes(tags)
        """)
        conn.commit()
        conn.close()

    def create_note(self, project_id: str, title: str, content: str,
                    tags: str = '', created_by: str = ''):
        """Create new note"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute("""
            INSERT INTO notes (project_id, title, content, tags, created_by)
            VALUES (?, ?, ?, ?, ?)
        """, (project_id, title, content, tags, created_by))

        note_id = cursor.lastrowid
        conn.commit()
        conn.close()

        return note_id
```

```python

    def get_notes(self, project_id: str) -> List[Dict]:
        """Get all notes for a project"""
        conn = sqlite3.connect(self.db_path)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        cursor.execute("""
            SELECT * FROM notes
            WHERE project_id = ?
            ORDER BY created_at DESC
        """, (project_id,))

        notes = [dict(row) for row in cursor.fetchall()]
        conn.close()

        return notes

    def update_note(self, note_id: int, title: str = None,
                   content: str = None, tags: str = None):
        """Update existing note"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        updates = []
        params = []

        if title:
            updates.append("title = ?")
            params.append(title)
        if content:
            updates.append("content = ?")
            params.append(content)
        if tags is not None:
            updates.append("tags = ?")
            params.append(tags)

        updates.append("updated_at = CURRENT_TIMESTAMP")
        params.append(note_id)

        query = f"UPDATE notes SET {', '.join(updates)} WHERE id = ?"
        cursor.execute(query, params)

        conn.commit()
        conn.close()

    def delete_note(self, note_id: int):
        """Delete note"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute("DELETE FROM notes WHERE id = ?", (note_id,))
        conn.commit()
        conn.close()

    def search_notes(self, query: str, project_id: str = None) -> List[Dict]:
        """Search notes by keyword"""
        conn = sqlite3.connect(self.db_path)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        search_pattern = f"%{query}%"

        if project_id:
            cursor.execute("""
                SELECT * FROM notes
                WHERE project_id = ?
                AND (title LIKE ? OR content LIKE ? OR tags LIKE ?)
                ORDER BY created_at DESC
            """, (project_id, search_pattern, search_pattern, search_pattern))
        else:
            cursor.execute("""
                SELECT * FROM notes
                WHERE title LIKE ? OR content LIKE ? OR tags LIKE ?
                ORDER BY created_at DESC
            """, (search_pattern, search_pattern, search_pattern))

        notes = [dict(row) for row in cursor.fetchall()]
        conn.close()

        return notes

    def export_notes(self, project_id: str) -> str:
        """Export all notes for a project as formatted text"""
        notes = self.get_notes(project_id)

        output = f"Project Notes: {project_id}\n"
        output += "=" * 50 + "\n\n"

        for note in notes:
            output += f"Title: {note['title']}\n"
            output += f"Date: {note['created_at']}\n"
            output += f"Tags: {note['tags']}\n"
            output += f"Created by: {note['created_by']}\n"
            output += "-" * 50 + "\n"
            output += f"{note['content']}\n\n"

        return output
```

6.3 Stakeholder Communication Hub

Email Generator (`src/communication/email_generator.py`):

```
import webbrowser
import urllib.parse
from jinja2 import Template
from typing import List, Dict

class EmailGenerator:
    def __init__(self, oracle_manager):
        self.db = oracle_manager
        self.templates = self._load_templates()

    def _load_templates(self):
        """Load email templates"""
        return {
            'status_update': """
Subject: Project Status Update - {{ project_name }}

Hi Team,

Here's the latest status update for {{ project_name }}:

Project ID: {{ project_id }}
Current Status: {{ status }}
PM: {{ pm_name }}
Target Completion: {{ target_date }}

Recent Progress:
{{ recent_progress }}

Upcoming Milestones:
{{ upcoming_milestones }}

Risks/Issues:
{{ risks }}

Please let me know if you have any questions.

Best regards,
{{ pm_name }}
            """,
            'executive_summary': """
Subject: Executive Summary - {{ project_name }}
```

```

Leadership Team,

Executive summary for {{ project_name }}:

• Status: {{ status }}
• On Track: {{ on_track }}
• Budget Status: {{ budget_status }}
• Key Achievements: {{ achievements }}
• Next Steps: {{ next_steps }}

Full details available in the project tracker.

{{ pm_name }}
            """,
            'delay_notification': """
Subject: Project Delay Notification - {{ project_name }}

Team,

This is to inform you of a delay in {{ project_name }}.

Original Target: {{ original_date }}
New Target: {{ new_date }}
Delay Reason: {{ delay_reason }}

Mitigation Plan:
{{ mitigation_plan }}

I will provide daily updates until we're back on track.

{{ pm_name }}
            """
        }

    def generate_email(self, template_name: str, project_data: Dict,
                      recipients: List[str], cc: List[str] = None):
        """Generate email using template"""
        template = Template(self.templates.get(template_name, ''))
        body = template.render(**project_data)

        # Create mailto link
        to_list = ';'.join(recipients)
        cc_list = ';'.join(cc) if cc else ''

        mailto_params = {
            'subject': f"Project Update - {project_data.get('project_name', 'N/A')}",
            'body': body
        }

        if cc_list:
            mailto_params['cc'] = cc_list

        mailto_url = f"mailto:{to_list}?{urllib.parse.urlencode(mailto_params)}"

        return {
            'mailto_url': mailto_url,
            'body': body
        }

    def open_email_client(self, mailto_url: str):
        """Open default email client with pre-filled email"""
        try:
            webbrowser.open(mailto_url)
            return {"status": "success"}
        except Exception as e:
            return {"status": "error", "message": str(e)}

    def get_stakeholders(self, project_id: str) -> List[Dict]:
        """Get stakeholder list from database"""
        query = """
        SELECT DISTINCT
            h.FULL_NAME,
            h.EMAIL,
            h.JOB_TITLE,
            h.ORGANIZATION
        FROM GS_REF_HR_DETAILS h
        JOIN GS_WFM_NF_PROJECTS p ON h.EMPLOYEE_ID = p.PM_ID
        WHERE p.NFID = :project_id
        """

        return self.db.execute_query(query, {'project_id': project_id})
```

---

7. Advanced Project Management & Collaboration

7.1 Interactive Gantt Charts

Gantt Generator (`src/gantt/generator.py`):

```
from datetime import datetime, timedelta
from typing import List, Dict

class GanttGenerator:
    def __init__(self, oracle_manager):
        self.db = oracle_manager

    def get_gantt_data(self, project_ids: List[str] = None,
                       start_date: str = None, end_date: str = None):
        """Generate Gantt chart data in DHTMLX format"""
        # Build query
        where_clauses = []
        params = {}

        if project_ids:
            placeholders = ','.join([f':id{i}' for i in range(len(project_ids))])
            where_clauses.append(f"p.NFID IN ({placeholders})")
            params.update({f'id{i}': pid for i, pid in enumerate(project_ids)})

        if start_date:
            where_clauses.append("m.MILESTONE_DATE >= :start_date")
            params['start_date'] = start_date

        if end_date:
            where_clauses.append("m.MILESTONE_DATE <= :end_date")
            params['end_date'] = end_date

        where_str = ' AND '.join(where_clauses) if where_clauses else '1=1'

        query = f"""
        SELECT
            p.NFID as id,
            p.PROJECT_NAME as text,
            m.MILESTONE_DATE as start_date,
            m.MILESTONE_DATE + 7 as end_date,
            p.STATUS as status,
            m.MILESTONE_NAME as milestone,
            p.PM_NAME as owner,
            CASE
                WHEN p.STATUS = 'Completed' THEN 1.0
                WHEN p.STATUS = 'In Progress' THEN 0.5
                ELSE 0.0
            END as progress
        FROM GS_WFM_NF_PROJECTS p
        LEFT JOIN GS_WFM_NF_PRJ_MILESTONES m ON p.NFID = m.NFID
        WHERE {where_str}
        ORDER BY m.MILESTONE_DATE
        """

        tasks = self.db.execute_query(query, params)

        # Get dependencies
        dep_query = """
        SELECT
            NFID as source,
            DEPENDENCY_NFID as target,
            DEPENDENCY_TYPE as type
        FROM GSC_WFM_NFID_DEPENDENCY_SUMMARY
        WHERE NFID IN ({})
        """.format(','.join([f"'{pid}'" for pid in project_ids])) if project_ids else ""

        links = []
        if dep_query:
            links = self.db.execute_query(dep_query)

        # Format for DHTMLX Gantt
        gantt_data = {
            'data': self._format_tasks(tasks),
            'links': self._format_links(links)
        }

        return gantt_data

    def _format_tasks(self, tasks: List[Dict]) -> List[Dict]:
        """Format tasks for DHTMLX Gantt"""
        formatted = []

        for task in tasks:
            formatted.append({
                'id': task['id'],
                'text': task['text'],
                'start_date': self._format_date(task['start_date']),
                'end_date': self._format_date(task['end_date']),
                'duration': 7,  # days
                'progress': task.get('progress', 0),
                'owner': task.get('owner', ''),
                'status': task.get('status', ''),
                'milestone': task.get('milestone', '')
            })

        return formatted

    def _format_links(self, links: List[Dict]) -> List[Dict]:
        """Format dependency links for DHTMLX Gantt"""
        formatted = []

        for idx, link in enumerate(links):
            formatted.append({
                'id': idx + 1,
                'source': link['source'],
                'target': link['target'],
                'type': self._get_link_type(link.get('type', 'FS'))
            })

        return formatted

    def _get_link_type(self, dep_type: str) -> int:
        """Convert dependency type to DHTMLX format"""
        type_map = {
            'FS': 0,  # Finish to Start
            'SS': 1,  # Start to Start
            'FF': 2,  # Finish to Finish
            'SF': 3   # Start to Finish
        }
        return type_map.get(dep_type, 0)

    def _format_date(self, date_obj) -> str:
        """Format date for DHTMLX"""
        if isinstance(date_obj, str):
            return date_obj
        elif isinstance(date_obj, datetime):
            return date_obj.strftime('%Y-%m-%d %H:%M')
        else:
            return datetime.now().strftime('%Y-%m-%d %H:%M')
```

7.2 Team Collaboration Hub

Task Manager (`src/tasks/manager.py`):

```
import sqlite3
from datetime import datetime
from typing import List, Dict, Optional

class TaskManager:
    def __init__(self, db_path='pm_tracker_user.db'):
        self.db_path = db_path
        self._initialize_db()

    def _initialize_db(self):
        """Create tasks table"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS custom_tasks (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                project_id VARCHAR(50) NOT NULL,
                title VARCHAR(255) NOT NULL,
                description TEXT,
                assigned_to VARCHAR(100),
                assigned_by VARCHAR(100),
                status VARCHAR(50) DEFAULT 'Open',
                priority VARCHAR(20) DEFAULT 'Medium',
                due_date DATE,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                completed_at TIMESTAMP
            )
        """)
        conn.commit()
        conn.close()

    def create_task(self, project_id: str, title: str, description: str = '',
                   assigned_to: str = '', assigned_by: str = '',
                   priority: str = 'Medium', due_date: str = None):
        """Create new custom task"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute("""
            INSERT INTO custom_tasks
            (project_id, title, description, assigned_to, assigned_by, priority, due_date)
            VALUES (?, ?, ?, ?, ?, ?, ?)
        """, (project_id, title, description, assigned_to, assigned_by, priority, due_date))

        task_id = cursor.lastrowid
        conn.commit()
        conn.close()

        return task_id

    def get_tasks(self, project_id: str = None, assigned_to: str = None,
                 status: str = None) -> List[Dict]:
        """Get tasks with optional filters"""
        conn = sqlite3.connect(self.db_path)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        query = "SELECT * FROM custom_tasks WHERE 1=1"
        params = []

        if project_id:
            query += " AND project_id = ?"
            params.append(project_id)

        if assigned_to:
            query += " AND assigned_to = ?"
            params.append(assigned_to)

        if status:
            query += " AND status = ?"
            params.append(status)

        query += " ORDER BY due_date ASC, priority DESC"

        cursor.execute(query, params)
        tasks = [dict(row) for row in cursor.fetchall()]
        conn.close()

        return tasks

    def update_task_status(self, task_id: int, status: str):
        """Update task status"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        completed_at = datetime.now() if status == 'Completed' else None

        cursor.execute("""
            UPDATE custom_tasks
            SET status = ?, completed_at = ?
            WHERE id = ?
        """, (status, completed_at, task_id))

        conn.commit()
        conn.close()

    def delete_task(self, task_id: int):
        """Delete task"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute("DELETE FROM custom_tasks WHERE id = ?", (task_id,))
        conn.commit()
        conn.close()
```

Comment System (`src/collaboration/comment_system.py`):

```
import sqlite3
import re
from datetime import datetime
from typing import List, Dict
from utils.window_manager import WindowManager

class CommentSystem:
    def __init__(self, db_path='pm_tracker_user.db'):
        self.db_path = db_path
        self.window_manager = WindowManager()
        self._initialize_db()

    def _initialize_db(self):
        """Create comments table"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS comments (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                project_id VARCHAR(50) NOT NULL,
                user_name VARCHAR(100) NOT NULL,
                user_email VARCHAR(255),
                comment_text TEXT NOT NULL,
                mentions TEXT,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
        """)
        conn.commit()
        conn.close()

    def add_comment(self, project_id: str, user_name: str,
                   user_email: str, comment_text: str):
        """Add new comment and process mentions"""
        # Extract mentions (@username)
        mentions = self._extract_mentions(comment_text)
        mentions_str = ','.join(mentions) if mentions else None

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute("""
            INSERT INTO comments
            (project_id, user_name, user_email, comment_text, mentions)
            VALUES (?, ?, ?, ?, ?)
        """, (project_id, user_name, user_email, comment_text, mentions_str))

        comment_id = cursor.lastrowid
        conn.commit()
        conn.close()

        # Send notifications for mentions
        if mentions:
            self._notify_mentions(mentions, user_name, project_id, comment_text)

        return comment_id

    def _extract_mentions(self, text: str) -> List[str]:
        """Extract @mentions from text"""
        pattern = r'@(\w+)'
        return re.findall(pattern, text)

    def _notify_mentions(self, mentions: List[str], from_user: str,
                        project_id: str, comment: str):
        """Send notifications to mentioned users"""
        for mention in mentions:
            self.window_manager.show_notification(
                f"Mentioned by {from_user}",
                f"In project {project_id}: {comment[:50]}..."
            )

    def get_comments(self, project_id: str) -> List[Dict]:
        """Get all comments for a project"""
        conn = sqlite3.connect(self.db_path)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        cursor.execute("""
            SELECT * FROM comments
            WHERE project_id = ?
            ORDER BY created_at DESC
        """, (project_id,))

        comments = [dict(row) for row in cursor.fetchall()]
        conn.close()

        return comments
```

7.3 Enterprise Tool Integration

Slack Integration (`src/integrations/slack.py`):

```
import webbrowser
import pyperclip
from typing import Dict

class SlackIntegration:
    def __init__(self, sqlite_manager):
        self.db = sqlite_manager

    def share_to_slack(self, project_data: Dict):
        """Generate Slack message and open Slack"""
        # Format message with Slack markdown
        message = f"""📊 *Project Update: {project_data['name']}*

*Status:* {project_data['status']}
*PM:* {project_data['pm_name']}
*Target Date:* {project_data.get('target_date', 'N/A')}
*Progress:* {project_data.get('progress', 0)}%

*Recent Updates:*
{project_data.get('recent_updates', 'No recent updates')}

*Next Steps:*
{project_data.get('next_steps', 'TBD')}
"""

        # Copy to clipboard
        pyperclip.copy(message)

        # Get associated Slack channel
        channel = self._get_project_channel(project_data['id'])

        # Open Slack
        if channel:
            slack_url = f"slack://channel?team=T1234&id={channel}"
        else:
            slack_url = "slack://open"

        webbrowser.open(slack_url)

        return {
            "status": "success",
            "message": "Message copied to clipboard. Slack opened.",
            "clipboard_content": message
        }

    def associate_channel(self, project_id: str, channel_id: str, channel_name: str):
        """Associate a Slack channel with a project"""
        conn = self.db.get_connection()
        cursor = conn.cursor()

        cursor.execute("""
            INSERT OR REPLACE INTO slack_channels
            (project_id, channel_id, channel_name)
            VALUES (?, ?, ?)
        """, (project_id, channel_id, channel_name))

        conn.commit()
        conn.close()

        return {"status": "success"}

    def _get_project_channel(self, project_id: str):
        """Get Slack channel ID for project"""
        conn = self.db.get_connection()
        cursor = conn.cursor()

        cursor.execute("""
            SELECT channel_id FROM slack_channels
            WHERE project_id = ?
        """, (project_id,))

        result = cursor.fetchone()
        conn.close()

        return result[0] if result else None

    def create_project_channel_link(self, project_name: str):
        """Generate a Slack channel creation URL"""
        # Slack doesn't support direct channel creation via URL
        # but we can prepare the name
        channel_name = project_name.lower().replace(' ', '-')[:80]

        return {
            "status": "info",
            "message": f"Suggested channel name: #{channel_name}",
            "channel_name": channel_name
        }
```

Webex Integration (`src/integrations/webex.py`):

```
import webbrowser
import urllib.parse
from datetime import datetime, timedelta
from typing import List, Dict

class WebexIntegration:
    def __init__(self, oracle_manager):
        self.db = oracle_manager

    def schedule_meeting(self, project_data: Dict, stakeholders: List[str],
                        meeting_duration: int = 60):
        """Schedule Webex meeting via Google Calendar"""
        # Calculate meeting time (next business day at 10 AM)
        meeting_time = self._get_next_business_day()
        end_time = meeting_time + timedelta(minutes=meeting_duration)

        # Format meeting details
        title = f"Project Review: {project_data['name']}"

        description = f"""Project: {project_data['name']}
Project ID: {project_data['id']}
PM: {project_data['pm_name']}

Agenda:
1. Project Status Review
2. Recent Accomplishments
3. Upcoming Milestones
4. Risks and Issues
5. Q&A

Webex Link: [Join Meeting]
"""

        # Build Google Calendar URL
        calendar_params = {
            'action': 'TEMPLATE',
            'text': title,
            'details': description,
            'dates': f"{meeting_time.strftime('%Y%m%dT%H%M%S')}/{end_time.strftime('%Y%m%dT%H%M%S')}",
            'add': ','.join(stakeholders) if stakeholders else ''
        }

        calendar_url = f"https://calendar.google.com/calendar/render?{urllib.parse.urlencode(calendar_params)}"

        # Open in browser
        webbrowser.open(calendar_url)

        return {
            "status": "success",
            "message": "Google Calendar opened with meeting details",
            "meeting_time": meeting_time.isoformat()
        }

    def _get_next_business_day(self):
        """Get next business day at 10 AM"""
        tomorrow = datetime.now() + timedelta(days=1)

        # Skip weekends
        while tomorrow.weekday() >= 5:  # 5 = Saturday, 6 = Sunday
            tomorrow += timedelta(days=1)

        # Set time to 10 AM
        return tomorrow.replace(hour=10, minute=0, second=0, microsecond=0)

    def generate_instant_meeting(self):
        """Generate instant Webex meeting link"""
        # Note: Without API, we can only open Webex
        webex_url = "webexteams://im"
        webbrowser.open(webex_url)

        return {
            "status": "success",
            "message": "Webex opened. Create meeting manually."
        }
```

Google Workspace Integration (`src/integrations/google_workspace.py`):

```
import webbrowser
import pyperclip
from typing import Dict

class GoogleWorkspaceIntegration:
    def __init__(self):
        pass

    def create_project_doc(self, project_data: Dict):
        """Create Google Doc with project template"""
        # Generate document content
        doc_content = f"""PROJECT BRIEF
{'=' * 80}

Project Name: {project_data['name']}
Project ID: {project_data['id']}
Project Manager: {project_data['pm_name']}
Status: {project_data['status']}
Created: {project_data.get('created_date', 'N/A')}
Target Completion: {project_data.get('target_date', 'N/A')}

{'=' * 80}

1. EXECUTIVE SUMMARY
   [Provide high-level overview of the project]

2. OBJECTIVES
   • Objective 1
   • Objective 2
   • Objective 3
```

```

3. SCOPE
   In Scope:
   •

   Out of Scope:
   •

4. STAKEHOLDERS
   {self._format_stakeholders(project_data.get('stakeholders', []))}

5. TIMELINE
   Key Milestones:
   {self._format_milestones(project_data.get('milestones', []))}

6. RISKS AND ISSUES
   [Document known risks and mitigation strategies]

7. SUCCESS CRITERIA
   [Define how success will be measured]

8. BUDGET
   [Include budget information if applicable]

{'=' * 80}
Document created by PM Project Tracker
"""

        # Copy template to clipboard
        pyperclip.copy(doc_content)

        # Open new Google Doc
        doc_url = "https://docs.google.com/document/create"
        webbrowser.open(doc_url)

        return {
            "status": "success",
            "message": "New Google Doc opened. Template copied to clipboard - paste it in.",
            "template": doc_content
        }

    def create_project_sheet(self, project_data: Dict):
        """Create Google Sheet for project tracking"""
        # Generate CSV data for sheet
        csv_data = f"""Task,Status,Owner,Due Date,Priority,Notes
Task 1,Not Started,{project_data['pm_name']},,Medium,
Task 2,Not Started,{project_data['pm_name']},,Medium,
Task 3,Not Started,{project_data['pm_name']},,Medium,
"""

        pyperclip.copy(csv_data)

        # Open new Google Sheet
        sheet_url = "https://docs.google.com/spreadsheets/create"
        webbrowser.open(sheet_url)

        return {
            "status": "success",
            "message": "New Google Sheet opened. Template data copied to clipboard.",
            "template": csv_data
        }

    def open_gmail_compose(self, project_data: Dict, recipients: List[str] = None):
        """Open Gmail compose with project info"""
        subject = f"Project Update - {project_data['name']}"
        body = f"""Hi Team,

Here's the latest update on {project_data['name']}:

Status: {project_data['status']}
PM: {project_data['pm_name']}

[Add your update here]

Best regards,
{project_data['pm_name']}
"""

        gmail_params = {
            'su': subject,
            'body': body
        }

        if recipients:
            gmail_params['to'] = ','.join(recipients)

        gmail_url = 
f"https://mail.google.com/mail/?view=cm&{urllib.parse.urlencode(gmail_params)}"
        webbrowser.open(gmail_url)

        return {
            "status": "success",
            "message": "Gmail compose opened"
        }

    def _format_stakeholders(self, stakeholders: List[Dict]) -> str:
        """Format stakeholder list"""
        if not stakeholders:
            return "   • No stakeholders defined"

        formatted = []
        for sh in stakeholders:
            formatted.append(f"   • {sh.get('name', 'Unknown')} - {sh.get('role', 'N/A')}")

        return '\n'.join(formatted)

    def _format_milestones(self, milestones: List[Dict]) -> str:
        """Format milestone list"""
        if not milestones:
            return "   • No milestones defined"

        formatted = []
        for ms in milestones:
            formatted.append(f"   • {ms.get('name', 'Unknown')} - {ms.get('date', 'TBD')}")

        return '\n'.join(formatted)
```

---

8. Intelligent Automation

8.1 Machine Learning Architecture

Delay Predictor (`src/ml/delay_predictor.py`):

```
import os
import numpy as np
import pandas as pd
from tensorflow import keras
from datetime import datetime

class DelayPredictor:
    def __init__(self, model_path='G:/Shared drives/TEAM-BAU/tracker/ml_models/delay_model.h5'):
        self.model_path = model_path
        self.model = self._load_model()
        self.feature_columns = [
            'project_duration', 'milestone_count', 'dependency_count',
            'team_size', 'complexity_score', 'pm_experience'
        ]

    def _load_model(self):
```

```python
        """Load trained model from G: drive"""
        if os.path.exists(self.model_path):
            return keras.models.load_model(self.model_path)
        else:
            print(f"Warning: Model not found at {self.model_path}")
            return None

    def predict_delay(self, project_data: dict) -> dict:
        """Predict if project will be delayed"""
        if not self.model:
            return {
                "status": "error",
                "message": "Model not loaded"
            }

        # Extract features
        features = self._extract_features(project_data)

        # Make prediction
        prediction = self.model.predict(features)
        delay_probability = float(prediction[0][0])

        # Calculate risk factors
        risk_factors = self._identify_risk_factors(project_data, delay_probability)

        return {
            "status": "success",
            "delay_probability": delay_probability,
            "is_at_risk": delay_probability > 0.7,
            "risk_level": self._get_risk_level(delay_probability),
            "risk_factors": risk_factors,
            "recommendations": self._get_recommendations(risk_factors)
        }

    def _extract_features(self, project_data: dict) -> np.ndarray:
        """Extract features for prediction"""
        # Calculate project duration
        start_date = datetime.strptime(project_data.get('start_date', '2025-01-01'), '%Y-%m-%d')
        target_date = datetime.strptime(project_data.get('target_date', '2025-12-31'), '%Y-%m-%d')
        duration_days = (target_date - start_date).days

        features = {
            'project_duration': duration_days,
            'milestone_count': project_data.get('milestone_count', 0),
            'dependency_count': project_data.get('dependency_count', 0),
            'team_size': project_data.get('team_size', 1),
            'complexity_score': project_data.get('complexity_score', 5),
            'pm_experience': project_data.get('pm_experience_years', 3)
        }

        # Convert to array
        feature_array = np.array([[features[col] for col in self.feature_columns]])

        return feature_array

    def _identify_risk_factors(self, project_data: dict, probability: float) -> list:
        """Identify contributing risk factors"""
        risk_factors = []

        if project_data.get('dependency_count', 0) > 5:
            risk_factors.append({
                "factor": "High Dependencies",
                "severity": "high",
                "description": "Project has many dependencies that could cause delays"
            })

        if project_data.get('milestone_count', 0) > 10:
            risk_factors.append({
                "factor": "Complex Timeline",
                "severity": "medium",
                "description": "Large number of milestones increases coordination complexity"
            })

        duration = (datetime.strptime(project_data.get('target_date', '2025-12-31'), '%Y-%m-%d') -
                   datetime.strptime(project_data.get('start_date', '2025-01-01'), '%Y-%m-%d')).days

        if duration > 180:
            risk_factors.append({
                "factor": "Long Duration",
                "severity": "medium",
                "description": "Extended timeline increases uncertainty"
            })

        if project_data.get('team_size', 1) < 3:
            risk_factors.append({
                "factor": "Small Team",
                "severity": "low",
                "description": "Limited resources may impact delivery"
            })

        return risk_factors
    def _get_risk_level(self, probability: float) -> str:
        """Convert probability to risk level"""
        if probability < 0.3:
            return "Low"
        elif probability < 0.6:
            return "Medium"
        elif probability < 0.8:
            return "High"
        else:
            return "Critical"

    def _get_recommendations(self, risk_factors: list) -> list:
        """Generate recommendations based on risk factors"""
        recommendations = []

        for factor in risk_factors:
            if factor['factor'] == "High Dependencies":
                recommendations.append("Review and document all dependencies. Create contingency plans.")
            elif factor['factor'] == "Complex Timeline":
                recommendations.append("Break down large milestones into smaller, manageable tasks.")
            elif factor['factor'] == "Long Duration":
                recommendations.append("Establish regular checkpoints and progress reviews.")
            elif factor['factor'] == "Small Team":
                recommendations.append("Consider additional resources or adjust scope.")

        if not recommendations:
            recommendations.append("Continue monitoring project metrics regularly.")

        return recommendations
```

Risk Classifier (`src/ml/risk_classifier.py`):

```
import os
import numpy as np
from tensorflow import keras

class RiskClassifier:
    def __init__(self, model_path='G:/Shared drives/TEAM-BAU/tracker/ml_models/risk_model.h5'):
        self.model_path = model_path
        self.model = self._load_model()
        self.risk_categories = ['Low', 'Medium', 'High', 'Critical']
    def _load_model(self):
        """Load trained model"""
        if os.path.exists(self.model_path):
            return keras.models.load_model(self.model_path)
        return None

    def classify_risk(self, project_data: dict) -> dict:
        """Classify project risk level"""
        if not self.model:
            return self._fallback_classification(project_data)

        # Extract features
        features = self._extract_features(project_data)

        # Predict risk category
        predictions = self.model.predict(features)
        risk_index = np.argmax(predictions[0])
        risk_level = self.risk_categories[risk_index]
        confidence = float(predictions[0][risk_index])

        return {
            "status": "success",
            "risk_level": risk_level,
            "confidence": confidence,
            "probabilities": {
                cat: float(prob)
                for cat, prob in zip(self.risk_categories, predictions[0])
            }
        }

    def _extract_features(self, project_data: dict) -> np.ndarray:
        """Extract features for classification"""
        # Similar to delay predictor
        features = np.array([[
            project_data.get('budget_variance', 0),
            project_data.get('schedule_variance', 0),
            project_data.get('resource_utilization', 100),
            project_data.get('stakeholder_satisfaction', 5),
            project_data.get('issue_count', 0)
        ]])

        return features

    def _fallback_classification(self, project_data: dict) -> dict:
        """Rule-based classification if model not available"""
        score = 0

        if project_data.get('status') == 'Delayed':
            score += 3
        if project_data.get('issue_count', 0) > 5:
            score += 2
        if project_data.get('budget_variance', 0) > 10:
            score += 2

        if score >= 5:
            risk_level = "Critical"
        elif score >= 3:
            risk_level = "High"
        elif score >= 1:
            risk_level = "Medium"
        else:
            risk_level = "Low"

        return {
            "status": "success",
            "risk_level": risk_level,
            "confidence": 0.8,
            "note": "Using rule-based classification"
        }
```

8.2 ML API Endpoints

`src/api/ml.py`:

```
from fastapi import APIRouter, HTTPException
from pydantic import BaseModel
from typing import Dict, Any
from ml.delay_predictor import DelayPredictor
from ml.risk_classifier import RiskClassifier

router = APIRouter()

# Initialize ML models
delay_predictor = DelayPredictor()
risk_classifier = RiskClassifier()

class PredictionRequest(BaseModel):
    project_data: Dict[str, Any]

@router.post("/predict/delay")
async def predict_delay(request: PredictionRequest):
    """Predict project delay"""
    try:
        result = delay_predictor.predict_delay(request.project_data)
        return result
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@router.post("/classify/risk")
async def classify_risk(request: PredictionRequest):
    """Classify project risk level"""
    try:
        result = risk_classifier.classify_risk(request.project_data)
        return result
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@router.get("/models/status")
async def get_models_status():
    """Get ML models status"""
    return {
        "delay_predictor": {
            "loaded": delay_predictor.model is not None,
            "path": delay_predictor.model_path
        },
        "risk_classifier": {
            "loaded": risk_classifier.model is not None,
            "path": risk_classifier.model_path
        }
    }
```

---

9. Deployment & Operations

9.1 G: Drive Structure for Deployment

```
G:\Shared drives\TEAM-BAU\tracker\
├── releases\
│   ├── PMTracker.exe          # Single executable
│   ├── version.json          # Version control
│   └── CHANGELOG.md              # Release notes
│
├── ml_models\
│   ├── delay_model.h5           # Delay prediction model
│   ├── risk_model.h5           # Risk classification model
│   └── model_metadata.json       # Model versions and info
│
├── docs\
│   ├── user_guide.pdf
│   ├── video_tutorials\
│   └── faq.md
│
├── templates\
│   ├── email_templates\
│   ├── report_templates\
│   └── note_templates\
│
└── logs\
    └── deployment_logs\
```

version.json format:

```
{
  "version": "10.1.0",
  "release_date": "2025-10-17",
  "checksum": "sha256:abc123...",
  "file_size": 85000000,
  "minimum_python": "3.11",
  "changes": [
    "Added PyWebView native desktop support",
    "Integrated text-to-speech functionality",
    "Enhanced integration with Slack, Webex, Google Workspace",
    "Improved ML model accuracy"
  ],
  "breaking_changes": false
}
```

9.2 Installation Process

For End Users:

1. Navigate to `G:\Shared drives\TEAM-BAU\tracker\releases\`
2. Copy `PMTracker.exe` to local folder (e.g., `C:\Apps\PMTracker\`)
3. Double-click `PMTracker.exe` to launch
4. First-time setup wizard appears:
   - Enter Oracle credentials
   - Test database connection
   - Configure preferences
5. Application opens in native window

First-Time Setup Wizard (`src/config/setup_wizard.py`):

```
import webview
from core.config_manager import ConfigManager
from core.oracle_manager import OracleManager

```

```
class SetupWizard:
    def __init__(self):
        self.config = ConfigManager()
        self.step = 1

    def run(self):
        """Run setup wizard"""
        html = self._generate_html()

        window = webview.create_window(
            'PM Tracker - First Time Setup',
            html=html,
            width=600,
            height=500,
            resizable=False,
            js_api=self
        )

        webview.start()

    def test_connection(self, username, password):
        """Test Oracle connection"""
        try:
            self.config.save_credentials(username, password)
            oracle = OracleManager(self.config.get_credentials())

            if oracle.test_connection():
                return {"status": "success", "message": "Connection successful!"}
            else:
                return {"status": "error", "message": "Connection failed"}
        except Exception as e:
            return {"status": "error", "message": str(e)}

    def save_and_finish(self, preferences):
        """Save preferences and complete setup"""
        # Save user preferences
        self.config.save_preferences(preferences)

        # Mark setup as complete
        self.config.set_setup_complete(True)

        return {"status": "success"}

    def _generate_html(self):
        """Generate setup wizard HTML"""
        return """
        <!DOCTYPE html>
        <html>
```

```
<head>
    <title>Setup Wizard</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            padding: 20px;
            background: #f5f5f5;
        }
        .container {
            background: white;
            padding: 30px;
            border-radius: 8px;
            box-shadow: 0 2px 10px rgba(0,0,0,0.1);
        }
        input {
            width: 100%;
            padding: 10px;
            margin: 10px 0;
            border: 1px solid #ddd;
            border-radius: 4px;
        }
        button {
            background: #007bff;
            color: white;
            padding: 10px 20px;
            border: none;
            border-radius: 4px;
            cursor: pointer;
        }
        button:hover {
            background: #0056b3;
        }
    </style>
</head>
<body>
    <div class="container">
        <h2>Welcome to PM Project Tracker</h2>
        <p>Please configure your Oracle database connection:</p>

        <label>Username:</label>
        <input type="text" id="username" value="splunkveep_nar" />

        <label>Password:</label>
        <input type="password" id="password" />

        <button onclick="testConnection()">Test Connection</button>
        <div id="result"></div>
    </div>
```

```

        <script>
            async function testConnection() {
                const username = document.getElementById('username').value;
                const password = document.getElementById('password').value;

                const result = await pywebview.api.test_connection(username, password);
                document.getElementById('result').innerHTML = result.message;

                if (result.status === 'success') {
                    setTimeout(() => window.close(), 2000);
                }
            }
        </script>
    </body>
    </html>
    """
```

9.3 Automatic Update Mechanism

Updater (`src/utils/updater.py`):

```
import os
import json
import hashlib
import shutil
import sys
import subprocess
from pathlib import Path

class Updater:
    def __init__(self, g_drive_path='G:/Shared drives/TEAM-BAU/tracker'):
        self.g_drive_path = g_drive_path
        self.current_version = "10.1.0"
        self.exe_name = "PMTracker.exe"

    def check_for_updates(self):
        """Check if update is available"""
        try:
            version_file = os.path.join(self.g_drive_path, 'releases', 'version.json')

            if not os.path.exists(version_file):
                return None

            with open(version_file, 'r') as f:
                latest_info = json.load(f)
```

```python

            if self._compare_versions(latest_info['version'], self.current_version) > 0:
                return latest_info

            return None
        except Exception as e:
            print(f"Error checking for updates: {e}")
            return None

    def _compare_versions(self, v1, v2):
        """Compare version strings"""
        v1_parts = [int(x) for x in v1.split('.')]
        v2_parts = [int(x) for x in v2.split('.')]

        for i in range(max(len(v1_parts), len(v2_parts))):
            part1 = v1_parts[i] if i < len(v1_parts) else 0
            part2 = v2_parts[i] if i < len(v2_parts) else 0

            if part1 > part2:
                return 1
            elif part1 < part2:
                return -1

        return 0

    def download_and_install(self, update_info):
        """Download and install update"""
        try:
            # Source and destination paths
            source_exe = os.path.join(self.g_drive_path, 'releases', self.exe_name)
            current_exe = sys.executable
            backup_exe = current_exe + '.backup'

            # Verify checksum
            if not self._verify_checksum(source_exe, update_info['checksum']):
                return {
                    "status": "error",
                    "message": "Checksum verification failed"
                }

            # Create backup
            shutil.copy(current_exe, backup_exe)

            # Copy new version
            shutil.copy(source_exe, current_exe)

            # Restart application
            subprocess.Popen([current_exe])
            sys.exit(0)

        except Exception as e:
            # Restore backup if something went wrong
            if os.path.exists(backup_exe):
                shutil.copy(backup_exe, current_exe)

            return {
                "status": "error",
                "message": str(e)
            }

    def _verify_checksum(self, file_path, expected_checksum):
        """Verify file checksum"""
        sha256_hash = hashlib.sha256()

        with open(file_path, 'rb') as f:
            for byte_block in iter(lambda: f.read(4096), b""):
                sha256_hash.update(byte_block)

        actual_checksum = f"sha256:{sha256_hash.hexdigest()}"
        return actual_checksum == expected_checksum

def check_for_updates():
    """Standalone function to check for updates"""
    updater = Updater()
    return updater.check_for_updates()
```

---

10. Development Setup

10.1 Prerequisites

- Windows 10/11
- Python 3.11+
- Git
- Visual Studio Code (recommended)
- Access to Verizon network and Oracle database

10.2 Environment Setup

```
# 1. Clone repository
git clone <repository_url>
cd PMTracker

# 2. Create virtual environment
```

```
python -m venv venv

# 3. Activate environment
venv\Scripts\activate

# 4. Upgrade pip
python -m pip install --upgrade pip

# 5. Install dependencies
pip install -r requirements.txt

# 6. Copy config template
copy config\config.ini.template config\config.ini

# 7. Edit config.ini with your credentials
notepad config\config.ini

# 8. Initialize local database
python scripts\init_db.py

# 9. Run application in development mode
python src\main.py
```

10.3 Building Executables

PyInstaller Spec File (`build.spec`):

```
# -*- mode: python ; coding: utf-8 -*-

block_cipher = None

# Analysis of dependencies
a = Analysis(
    ['src/main.py'],
    pathex=[],
    binaries=[],
    datas=[
        ('src/web_app/static', 'web_app/static'),
        ('src/web_app/templates', 'web_app/templates'),
        ('resources', 'resources'),
        ('config/logging.yaml', 'config'),
    ],
    hiddenimports=[
        'uvicorn.logging',
        'uvicorn.loops',
        'uvicorn.loops.auto',
        'uvicorn.protocols',
        'uvicorn.protocols.http',
        'uvicorn.protocols.http.auto',
        'uvicorn.protocols.websockets',
        'uvicorn.protocols.websockets.auto',
        'uvicorn.lifespan',
        'uvicorn.lifespan.on',
        'pydantic',
        'oracledb',
        'cryptography',
        'gtts',
        'pydub',
        'tensorflow',
        'keras',
    ],
    hookspath=[],
    hooksconfig={},
    runtime_hooks=[],
    excludes=[
        'matplotlib',
        'tkinter',
    ],
    win_no_prefer_redirects=False,
    win_private_assemblies=False,
    cipher=block_cipher,
    noarchive=False,
)

# Package everything
pyz = PYZ(a.pure, a.zipped_data, cipher=block_cipher)

exe = EXE(
    pyz,
    a.scripts,
    a.binaries,
    a.zipfiles,
    a.datas,
    [],
    name='PMTracker',
    debug=False,
    bootloader_ignore_signals=False,
    strip=False,
    upx=True,
    upx_exclude=[],
    runtime_tmpdir=None,
    console=False,  # No console window
    disable_windowed_traceback=False,
    target_arch=None,
    codesign_identity=None,
    entitlements_file=None,
    icon='resources/icon.ico',
    version='version_info.txt'
)
```

Version Info File (`version_info.txt`):

```
VSVersionInfo(
  ffi=FixedFileInfo(
    filevers=(10, 1, 0, 0),
    prodvers=(10, 1, 0, 0),
    mask=0x3f,
    flags=0x0,
    OS=0x40004,
    fileType=0x1,
    subtype=0x0,
    date=(0, 0)
  ),
  kids=[
    StringFileInfo(
      [
      StringTable(
        u'040904B0',
        [StringStruct(u'CompanyName', u'Verizon Internal Systems'),
        StringStruct(u'FileDescription', u'PM Project Tracker - Native Desktop Application'),
        StringStruct(u'FileVersion', u'10.1.0.0'),
        StringStruct(u'InternalName', u'PMTracker'),
        StringStruct(u'LegalCopyright', u'Copyright (c) 2025 Verizon'),
        StringStruct(u'OriginalFilename', u'PMTracker.exe'),
        StringStruct(u'ProductName', u'PM Project Tracker'),
        StringStruct(u'ProductVersion', u'10.1.0.0')])
      ]
    ),
    VarFileInfo([VarStruct(u'Translation', [1033, 1200])])
  ]
)
```

Build Script (`scripts/build.py`):

```
import os
import subprocess
import shutil
from datetime import datetime

def clean_build():
    """Clean previous build artifacts"""
    dirs_to_remove = ['build', 'dist']
    for dir_name in dirs_to_remove:
        if os.path.exists(dir_name):
            print(f"Removing {dir_name}...")
            shutil.rmtree(dir_name)

def build_executable():
    """Build executable using PyInstaller"""
    print("Building PMTracker.exe...")
    result = subprocess.run(['pyinstaller', 'build.spec'], capture_output=True)

    if result.returncode == 0:
        print("Build successful!")
        return True
    else:
        print("Build failed!")
        print(result.stderr.decode())
        return False

def copy_to_gdrive(g_drive_path='G:/Shared drives/TEAM-BAU/tracker/releases'):
    """Copy executable to G: drive"""
    if not os.path.exists(g_drive_path):
        print(f"G: drive path not found: {g_drive_path}")
        return False

    source = 'dist/PMTracker.exe'
    destination = os.path.join(g_drive_path, 'PMTracker.exe')

    print(f"Copying to {destination}...")
    shutil.copy(source, destination)

    print("Copy successful!")
    return True

def update_version_json(g_drive_path='G:/Shared drives/TEAM-BAU/tracker/releases'):
    """Update version.json file"""
    import json
    import hashlib

    version_file = os.path.join(g_drive_path, 'version.json')
    exe_file = os.path.join(g_drive_path, 'PMTracker.exe')

    # Calculate checksum
    sha256_hash = hashlib.sha256()
    with open(exe_file, 'rb') as f:
        for byte_block in iter(lambda: f.read(4096), b""):
            sha256_hash.update(byte_block)

    checksum = f"sha256:{sha256_hash.hexdigest()}"
    file_size = os.path.getsize(exe_file)

    # Create version info
    version_info = {
        "version": "10.1.0",
        "release_date": datetime.now().strftime("%Y-%m-%d"),
        "checksum": checksum,
        "file_size": file_size,
        "minimum_python": "3.11",
        "changes": [
            "PyWebView native desktop support",
            "Text-to-speech functionality",
            "Enhanced integrations",
            "Improved ML models"
        ],
        "breaking_changes": False
    }

    with open(version_file, 'w') as f:
        json.dump(version_info, f, indent=2)

    print("version.json updated!")

if __name__ == '__main__':
    print("=" * 50)
    print("PM Tracker Build Script")
    print("=" * 50)

    # Clean previous builds
    clean_build()

    # Build executable
    if build_executable():
        # Copy to G: drive
        if copy_to_gdrive():
            # Update version info
            update_version_json()
            print("\nBuild and deployment complete!")
        else:
            print("\nBuild complete, but deployment to G: drive failed.")
    else:
        print("\nBuild failed!")
```

Usage:

```
# Build executable
python scripts/build.py

# Or manually with PyInstaller
pyinstaller build.spec
```

10.4 Testing

Unit Tests (`tests/test_api.py`):

```
import pytest
from fastapi.testclient import TestClient
from api import create_app

@pytest.fixture
def client():
    app = create_app()
    return TestClient(app)

def test_health_check(client):
    """Test health check endpoint"""
    response = client.get("/api/health")
    assert response.status_code == 200
    assert response.json()["status"] == "healthy"

def test_projects_endpoint(client):
    """Test projects list endpoint"""
    response = client.get("/api/projects")
    assert response.status_code == 200
    data = response.json()
    assert "projects" in data

def test_ml_models_status(client):
    """Test ML models status endpoint"""
    response = client.get("/api/ml/models/status")
    assert response.status_code == 200
    data = response.json()
    assert "delay_predictor" in data
    assert "risk_classifier" in data
```

Run Tests:

```
# Run all tests
```

```
pytest

# Run with coverage
pytest --cov=src tests/

# Run specific test file
pytest tests/test_api.py
```

---

11. Appendices

Appendix A: GS Naming Conventions

The Golden Source schema follows strict naming conventions:

Tables:

- Format: `GS_<LAYER>_<TABLE_NAME>`
- Example: `GS_WFM_NF_PROJECTS`

Views:

- Format: `GSV_<LAYER>_<VIEW_NAME>`
- Example: `GSV_REF_HR_DETAILS_CURRENT`

Calculated Tables:

- Format: `GSC_<LAYER>_<TABLE_NAME>`
- Example: `GSC_WFM_UTE_TASKS`

Indexes:

- Format: `IDX_<TABLE>_<COLUMN>`
- Example: `IDX_PROJECTS_STATUS`

Appendix B: Detailed Data Layers

Layer	Description	Key Tables	Refresh Frequency
ADMIN	GS administration and access control	GS_ADMIN_USERS, GS_ADMIN_LOGS	Real-time

AYS	AYS ticketing system data	GS_AYS_TICKETS, GS_AYS_ASSIGNMENTS	Daily
CCP	CCP system (CCRs, TEOs, Queues)	GS_CCP_CCRS, GS_CCP_TEOS, GS_CCP_CCR_QUEUES	Daily
DECOM	Decommissioning program data	GS_DECOM_ASSETS, GS_DECOM_SITES	Weekly
ETI	ETI team dashboards and reports	GS_ETI_METRICS, GS_ETI_REPORTS	Daily
NAUT	NAUTILIS system data	GS_NAUT_ORDERS	Daily
PMRA	PMRA order system	GS_PMRA_PROV_ORDERS	Daily
PORCH	PORCH customer orders	GS_PORCH_CUSTOMER_ORDERS	Daily
REF	Reference data (HR, calendars)	GS_REF_HR_DETAILS, GS_REF_WORKDAY_CALENDAR	Daily
WFM	Canvas WFM (NFIDs, tasks)	GS_WFM_NF_PROJECTS, GS_WFM_UTE_TASKS	Daily

Appendix C: Key ETL Processes

ETL Script	Target Tables	Description	Dependencies
`Refresher_CCP_CCRs.py`	GS_CCP_CCRS, GS_CCP_CCR_VOLUME	Core CCR dat	None

		a and volume metrics	
Refresher_PMRA_Prov_Orders.py	GS_PMRA_PROV_ORDERS	Provisioning order data	REF layer
Refresher_WFM_NFID_Dependency.py	GSC_WFM_NFID_DEPENDENCY_SUMMARY	WFM project and tas	WFM layer

		k data	
`Refresher_PORCH_Customer_Orders.py`	GS_PORCH_CUSTOMER_ORDERS	Customer order details	REF layer
`Refresher_AYS_Tickets.py`	GS_AYS_TICKETS	Ticket data via API	REF layer
`Refresher_STG_HR.py`	GS_REF_HR_DETAILS	Employee and orgh	None

i
e
r
a
r
c
h
y

Appendix D: API Reference

Projects API

2524. GET /api/projects - List all projects (with filters)
2525. GET /api/projects/{id} - Get project details
2526. POST /api/projects/{id}/notes - Add project note
2527. GET /api/projects/{id}/tasks - Get project tasks

Reports API

2528. POST /api/reports/build - Build custom report
2529. POST /api/reports/export - Export report (PDF/Excel)
2530. GET /api/reports/performance - Performance summary

Gantt API

2531. GET /api/gantt/data - Get Gantt chart data
2532. POST /api/gantt/dependencies - Update dependencies

Tasks API

2533. GET /api/tasks - List tasks (filtered)
2534. POST /api/tasks - Create new task
2535. PUT /api/tasks/{id} - Update task
2536. DELETE /api/tasks/{id} - Delete task

Notes API

2537. GET /api/notes/{project_id} - Get project notes
2538. POST /api/notes - Create note
2539. PUT /api/notes/{id} - Update note
2540. DELETE /api/notes/{id} - Delete note
2541. GET /api/notes/search - Search notes

ML API

2542. POST /api/ml/predict/delay - Predict project delay
2543. POST /api/ml/classify/risk - Classify project risk

```
GET    /api/ml/models/status      - Get ML models status
```

TTS API

```
POST   /api/tts/speak           - Text-to-speech
POST   /api/tts/stop            - Stop speaking
GET    /api/tts/status          - Get TTS status
```

Appendix E: Glossary

Term	Definition
Airflow	Open-source platform for workflow orchestration
CCR	Customer Carrier Request
CLLI	Common Language Location Identifier
DAG	Directed Acyclic Graph (Airflow workflow)
ETL	Extract, Transform, Load
GS	Golden Source (centralized data repository)
KPI	Key Performance Indicator
NFID	Network Facility Identifier
PMRA	Project Management and Reporting Application
PyWebView	Python library for native desktop windows
TEO	Telecommunications Equipment Order
TTS	Text-to-Speech
UTE	Universal Tasking Engine
WFM	Workflow Management (Canvas)

Appendix F: Frontend JavaScript Examples

Dashboard Widget (`web_app/static/js/dashboard.js`):

```
// Initialize dashboard
class Dashboard {
```

```
constructor() {
    this.grid = null;
    this.widgets = [];
    this.init();
}

init() {
    // Initialize GridStack
    this.grid = GridStack.init({
        cellHeight: 80,
        acceptWidgets: true,
        removable: '.trash'
    });

    // Load saved layout
    this.loadLayout();

    // Setup event listeners
    this.setupEventListeners();
}

async loadLayout() {
    try {
        const response = await fetch('/api/dashboard/layout');
        const layout = await response.json();

        if (layout.widgets) {
            layout.widgets.forEach(widget => {
                this.addWidget(widget);
            });
        }
    } catch (error) {
        console.error('Error loading layout:', error);
    }
}

addWidget(config) {
    const widget = document.createElement('div');
    widget.className = 'grid-stack-item';
    widget.setAttribute('gs-x', config.x);
    widget.setAttribute('gs-y', config.y);
    widget.setAttribute('gs-w', config.w);
    widget.setAttribute('gs-h', config.h);

    const content = this.renderWidget(config.type);
    widget.innerHTML = `
        <div class="grid-stack-item-content">
            ${content}
        </div>
    `;

    this.grid.addWidget(widget);
    this.widgets.push(config);
}

renderWidget(type) {
    switch(type) {
        case 'kpi':
            return this.renderKPI();
        case 'projects':
            return this.renderProjectList();
        case 'tasks':
            return this.renderTaskList();
        default:
            return '<div>Unknown widget</div>';
    }
}

renderKPI() {
    return `
        <div class="widget-header">
            <h3>Key Metrics</h3>
        </div>
        <div class="widget-body">
            <div class="kpi-grid">
                <div class="kpi-card">
                    <div class="kpi-value" id="total-projects">0</div>
                    <div class="kpi-label">Total Projects</div>
                </div>
                <div class="kpi-card">
                    <div class="kpi-value" id="active-projects">0</div>
                    <div class="kpi-label">Active</div>
                </div>
            </div>
        </div>
    `;
}

async updateKPIs() {
    try {
        const response = await fetch('/api/projects/kpis');
        const data = await response.json();

        document.getElementById('total-projects').textContent = data.total;
        document.getElementById('active-projects').textContent = data.active;
    } catch (error) {
```

```javascript
            console.error('Error updating KPIs:', error);
        }
    }

    setupEventListeners() {
        // Save layout on change
        this.grid.on('change', () => {
            this.saveLayout();
        });

        // Update data periodically
        setInterval(() => {
            this.updateKPIs();
        }, 60000); // Every minute
    }

    async saveLayout() {
        const layout = {
            widgets: this.grid.save()
        };

        try {
            await fetch('/api/dashboard/layout', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify(layout)
            });
        } catch (error) {
            console.error('Error saving layout:', error);
        }
    }
}

// Initialize on page load
document.addEventListener('DOMContentLoaded', () => {
    const dashboard = new Dashboard();
});
```

TTS Integration (`web_app/static/js/tts.js`):

```javascript
class TTSController {
    constructor() {
        this.isSpeaking = false;
        this.setupButtons();
    }

    setupButtons() {
        // Add speak button to project updates
        document.querySelectorAll('.project-update').forEach(element => {
            const speakBtn = document.createElement('button');
            speakBtn.className = 'btn-speak';
            speakBtn.innerHTML = '<i data-lucide="volume-2"></i>';
            speakBtn.addEventListener('click', () => {
                this.speak(element.textContent);
            });
            element.appendChild(speakBtn);
        });
    }

    async speak(text) {
        if (this.isSpeaking) {
            await this.stop();
            return;
        }

        try {
            const response = await fetch('/api/tts/speak', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({
                    text: text,
                    language: 'en',
                    slow: false
                })
            });

            const result = await response.json();

            if (result.status === 'speaking') {
                this.isSpeaking = true;
                this.updateUI();
            }
        } catch (error) {
            console.error('TTS Error:', error);
        }
    }

    async stop() {
        try {
            await fetch('/api/tts/stop', { method: 'POST' });
            this.isSpeaking = false;
            this.updateUI();
        } catch (error) {
            console.error('Error stopping TTS:', error);
        }
    }

    updateUI() {
        document.querySelectorAll('.btn-speak').forEach(btn => {
            if (this.isSpeaking) {
                btn.innerHTML = '<i data-lucide="volume-x"></i>';
                btn.classList.add('speaking');
            } else {
                btn.innerHTML = '<i data-lucide="volume-2"></i>';
                btn.classList.remove('speaking');
            }
        });
    }
}

// Initialize TTS controller
const ttsController = new TTSController();
```

Appendix G: Troubleshooting Guide

Issue	Possible Cause	Solution
Application won't start	Port 8000 already in use	Close other apps using port 8000
Database connection fails	Incorrect credentials	Run setup wizard again
White screen on launch	Frontend files missing	Rebuild application with all assets
TTS not working	Audio output not available	Check sound settings
Auto-update fails	G: drive not accessible	Check VPN/network connection
ML predictions fail	Models not found on G: drive	Verify G: drive path and model files
Slow performance	Too many widgets on dashboard	Remove unused widgets
Gantt chart not loading	No project data	Verify database query returns data

Appendix H: Performance Optimization Tips

1. **Database Queries**
   - Use connection pooling (already implemented)
   - Add indexes for frequently queried columns
   - Limit result sets with WHERE clauses
2. **Frontend Performance**
   - Lazy load widgets
   - Use pagination for large lists
   - Implement virtual scrolling
3. **Memory Management**
   - Close database connections properly
   - Clear cached data periodically
   - Limit ML model memory usage
4. **Network Optimization**
   - Cache static assets
   - Compress API responses
   - Use WebSocket for real-time updates

Conclusion

This PyWebView-based PM Project Tracker provides a **comprehensive, native desktop solution** that:

✅ **Works within your constraints:**

- No external browser required
- No API keys needed
- No additional software installation
- Single executable deployment

✅ **Delivers enterprise features:**

- Direct Oracle database integration
- Advanced reporting and analytics
- Interactive Gantt charts
- Team collaboration tools
- ML-powered predictions
- Text-to-speech accessibility

✅ **Provides seamless integrations:**

- Slack (message formatting + deep links)
- Webex (calendar integration)
- Google Workspace (Docs, Sheets, Gmail)
- All via desktop automation and clipboard

✅ **Ensures easy maintenance:**

- Auto-update mechanism
- Centralized deployment via G: drive
- Modular architecture
- Comprehensive logging