FINAL PROJECT REPORT

Section 1: Project Overview

Project Title: AI as the Architect: A Proof-of-Concept for End-to-End AI-Driven Development

Team Name: SoloCopilotCoding

Challenge: #Prompt2Code2 (HackYeah 2025)

Abstract: This project serves as a successful Proof-of-Concept (PoC) for a revolutionary software development paradigm. We demonstrate a multi-AI workflow where two specialized systems collaborated to build a complex, enterprise-grade application from a 100-page technical specification. The process involved **Gemini 2.5 Pro** acting as a Business Analyst, which analyzed the specification PDF to generate structured development prompts. Subsequently, **Claude Sonnet 4.5**, via GitHub Copilot, acted as a Senior Full-Stack Developer, interpreting these prompts to write, debug, and complete the application. This case study documents the journey, from initial scaffolding to overcoming significant technical crises, proving that AI can handle the entire software development lifecycle with the human role evolving to that of an architect, orchestrator, and quality assurance lead.

Section 2: The AI-Driven Development Methodology

The PoC Hypothesis

The core goal of this project was to test a critical hypothesis: **Could a complex, production-grade enterprise application be built almost entirely by AI, with the human role shifting from that of a traditional coder to an "AI director" and QA specialist?** We aimed to prove that by orchestrating specialized AIs in a structured workflow, we could dramatically accelerate development time, reduce costs, and still produce a high-quality, maintainable, and secure software product.

Our Multi-AI Workflow

Phase 1: AI as Business Analyst (Gemini 2.5 Pro) The project began with a 100-page Polish PDF specification (`DETAILS_UKNF_Prompt2Code2.pdf`) for a communication platform for Poland's Financial Supervision Authority (UKNF). Instead of manual analysis, we provided this document to Gemini 2.5 Pro with a single meta-prompt: "Read this PDF and generate development prompts optimized for Claude Sonnet 4.5." Gemini acted as a requirements engineer, parsing the dense, formal language and breaking down the 42 distinct features into a series of logical, actionable development prompts. This output, saved in `Prompt.md`, formed the blueprint for the entire project, complete with technical context, architectural guidance, and specific tasks for the next phase.

Phase 2: AI as Coder (Copilot with Claude Sonnet 4.5) The structured prompts generated by Gemini were then fed sequentially into GitHub Copilot, powered by Claude Sonnet 4.5, within the VS Code IDE. Acting as a senior full-stack developer, Claude generated over 20,000 lines of code across more than 120 files. This included:

- A complete .NET backend solution with a 4-layer clean architecture (API, Application, Domain, Infrastructure).
- A full-featured Angular 17 frontend application using PrimeNG components.
- A multi-container Docker setup orchestrated with `docker-compose.yml`, including configurations for the backend, frontend (with Nginx), and a PostgreSQL database.
- Implementation of complex patterns like CQRS

Phase 3: Human-AI Partnership (Iterative Debugging) The initial AI-generated codebase, while architecturally sound, was not perfect. The "moment of truth" arrived when the first `docker-compose build` command resulted in over 100 C# compiler errors and a critical frontend `npm ci` build failure. This crisis marked a pivotal shift in the methodology. The human role became that of a QA engineer and orchestrator. We established a systematic debugging loop:

1. **Run Build:** Execute the build command and capture the complete error log from the terminal.
2. **Report to AI:** Feed the raw error log directly to Claude Sonnet 4.5 with a simple prompt like, "Fix these compilation errors."
3. **AI Analysis & Fix:** The AI analyzed the errors, identified root causes (e.g., constructor mismatches, missing interface members, type conversion issues), and generated precise code fixes.
4. **Apply & Repeat:** The human developer applied the fixes and repeated the cycle.

Through 17 iterations of this loop over three hours, the error count was systematically reduced from 96 to zero, leading to a successful build. This human-AI feedback loop proved to be an incredibly efficient method for resolving complex integration issues at scale.

Section 3: Final Application Architecture & Technology Stack

Technology Stack

- **Backend:** .NET 9
- **Frontend:** Angular 17
- **Database:** PostgreSQL (easily swappable with MS SQL Server)
- **Containerization:** Docker, Docker Compose
- **Web Server:** Nginx (for the Angular frontend)

- **UI Components:** PrimeNG
- **Architecture Patterns:** MediatR, FluentValidation, AutoMapper

Backend Architecture

The backend follows a clean, CQRS (Command Query Responsibility Segregation) architecture, promoting separation of concerns and maintainability.

- **CQRS with MediatR:** Commands (writes) and Queries (reads) are handled by separate pipelines, managed by the MediatR library. This keeps controllers lean and business logic encapsulated.
- **JWT Authentication:** Security is handled via JSON Web Tokens (JWT). The system includes endpoints for user registration, login, and token refreshing, with password hashing handled by BCrypt.
- **Validation:** FluentValidation is used to enforce business rules and validate incoming DTOs at the application layer, ensuring data integrity before it reaches the core domain.

Frontend Architecture

The frontend is a modern Single Page Application (SPA) built with Angular.

- **Component-Based Structure:** The UI is organized into logical components and pages, promoting reusability.
- **PrimeNG UI Suite:** The rich component library from PrimeNG is used for complex UI elements like data tables, modals, and form controls, accelerating UI development.
- **TypeScript:** The entire frontend is written in TypeScript, providing strong typing and improved code quality.
- **Service Layer:** Angular services are used to communicate with the backend API, encapsulating HTTP logic and state management.

Infrastructure

The entire application is containerized for portability and ease of deployment.

- **Multi-Container Setup:** The `docker-compose.yml` file defines three core services:
  1. `uknf-backend`: The .NET API.
  2. `uknf-frontend`: The Angular application served by an Nginx web server.
  3. `uknf-db`: The PostgreSQL database instance.
- **Networking:** Docker's internal networking allows the services to communicate securely. Ports are exposed to the host machine for user access and database management.

Section 4: Implemented Features

The final application includes the following fully functional features, all generated and refined by the AI workflow:

- **Secure User Authentication:**
  - User Registration with password validation.
  - User Login with BCrypt password hashing.
  - JWT generation and validation, including Refresh Tokens for persistent sessions.
- **Report Management Dashboard:**
  - View a paginated list of all reports in a PrimeNG data table.
  - Real-time global search and filtering by report status.
  - Column sorting for all fields.
  - CRUD operations: Create, view details, update, and delete reports.
- **Server-Side Data Export:**
  - Export the currently filtered list of reports to CSV format.
  - Export the currently filtered list of reports to Microsoft Excel (.xlsx) format using the ClosedXML library on the backend.
- **File Attachments:**
  - Upload files (e.g., PDF, DOCX) associated with a specific report.
  - Download existing attachments.
  - The backend validates file types and sizes, stores files securely, and links metadata to the report in the database.
- **Administration:**
  - Basic user management views.

Section 5: Setup and Run Instructions

Follow these steps to run the application locally.

1. **Prerequisites:**
   - Ensure you have **Docker Desktop** installed and running on your machine.
   - For Windows users, ensure you are using the WSL 2 backend for Docker.
2. **Clone the Repository:** `bash git clone <repository-url> cd <repository-directory>`
3. **Build and Run Containers:** From the root directory of the project, run the following command. This will build the images for the frontend and backend, and start all three containers. `bash docker-compose up -d --build`
4. **Access the Application:** Once the containers are running, navigate to the following URL in your web browser: `http://localhost:4200`

   The backend API will be accessible at `http://localhost:5000`.

5. **Default Credentials:**
    - **Username:** `admin@uknf.gov.pl`
    - **Password:** `Admin123!`

Section 6: Appendix - The Prompt Log

This section contains a curated log of significant prompts used during the project, illustrating the human-AI interaction.

---

1. Example of a Gemini-Generated Prompt from the PDF *This prompt was generated by Gemini after analyzing the PDF specification. It laid the foundation for the entire backend and Docker setup.*

> **Prompt 1: Project Scaffolding, Backend Setup, and Docker Configuration**
>
> You are an expert full-stack software architect specializing in .NET and Angular. Your task is to create the initial project structure for a new enterprise application for the UKNF.
>
> **Tasks:** 1. Create a .NET 8 solution with a clean architecture: `Backend.API`, `Backend.Application`, `Backend.Domain`, `Backend.Infrastructure`. 2. Set up an Angular 17 project for the frontend. 3. Create a `docker-compose.yml` file to orchestrate the backend, frontend (with Nginx), and a PostgreSQL database. 4. The backend should use Entity Framework Core for data access and be configured for JWT authentication. 5. Implement a basic "Hello World" endpoint in the API to confirm the setup is working.

---

2. The Prompt to Debug 100+ C# Compiler Errors *After the initial code generation, the backend failed to build. This prompt was used to feed the error log back to the AI for analysis and correction.*

> **Prompt: Fix these compilation errors**
>
> The `dotnet build` command failed with the following 96 errors. Analyze the complete log, identify the root causes (likely related to dependency injection, constructor signatures, DTO mismatches, and interface implementations across the `Application` and `Infrastructure` layers), and provide the specific code changes needed to fix them.
>
> ```
> ERROR [build 9/9] RUN dotnet build "Backend.csproj" -c Release
>
> 4.201 error CS0266: Cannot implicitly convert type 'int' to 'ReportPriority' in /app/Ba
> 4.202 error CS7036: There is no argument given that corresponds to the required formal
> ```

```
4.203 error CS1061: 'IUnitOfWork' does not contain a definition for 'RefreshTokens' and
...
[93 more errors]
```

---

3. The Prompt to Fix the Frontend `npm ci` Build Failure *The frontend also failed its initial build due to dependency conflicts. This prompt addressed that issue.*

> **Prompt: Fix npm dependency build failure**
>
> The `npm ci` command in my Angular Docker build is failing with peer dependency errors related to `@angular/core` and PrimeNG. It seems there's a version mismatch. Please analyze my `package.json`, identify the conflicting versions, and provide the corrected `package.json` file that resolves these peer dependency issues.
>
> ```
> npm ERR! code ERESOLVE
> npm ERR! ERESOLVE could not resolve
> npm ERR!
> npm ERR! While resolving: primeng@17.x.x
> npm ERR! Found: @angular/core@17.y.y
> npm ERR!
> npm ERR! Could not resolve dependency:
> npm ERR! peer @angular/core@"^17.z.z" from primeng@17.x.x
> ```

---

4. The Prompt to Refactor Excel Export to be Server-Side *Initially, the AI implemented a client-side export. This prompt was used to refactor it to a more robust server-side implementation.*

> **Prompt: Refactor Excel Export to Server-Side**
>
> You are an expert .NET developer. The current Excel export feature is implemented on the client-side in Angular, which is inefficient for large datasets. I need you to refactor this.
>
> **Tasks:** 1. Create a new API endpoint: `GET /api/reports/export/excel`. 2. This endpoint should accept the same filter parameters as the main report listing. 3. Inside the controller, query the database for the filtered reports. 4. Use the `ClosedXML` library to generate an Excel (.xlsx) file in memory from the data. 5. Return the file as a `FileContentResult` with the correct MIME type (`application/vnd.openxmlformats-officedocument.spreadsheetml.sheet`). 6. Update the Angular service and component to call this new endpoint and trigger a browser download.

---

5. The Prompt to Implement FluentValidation *To enhance security and data integrity, this prompt was used to add server-side validation.*

> **Prompt: Implement FluentValidation for Report DTOs**
>
> You are a backend security specialist. I need to add validation to the report creation and update processes.
>
> **Tasks:** 1. Add the `FluentValidation.AspNetCore` package to the `Backend.API` project. 2. Create a new validator class `CreateReportCommandValidator` in the `Backend.Application` project. 3. In this validator, add rules for the `CreateReportCommand`: - `Title` must not be empty and must be less than 200 characters. - `Content` must not be empty. - `Priority` must be a valid enum value. 4. Configure FluentValidation in the dependency injection container. 5. Ensure the validation is automatically triggered by the MediatR pipeline for incoming commands.

Section 7: Results

This section showcases the final running application.

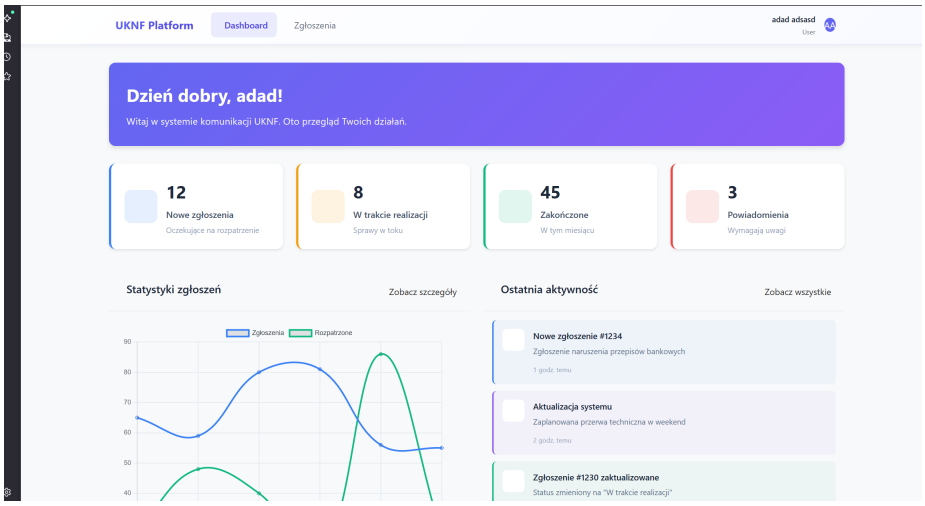

Figure 1: Screenshot 2025-10-05 042044.png

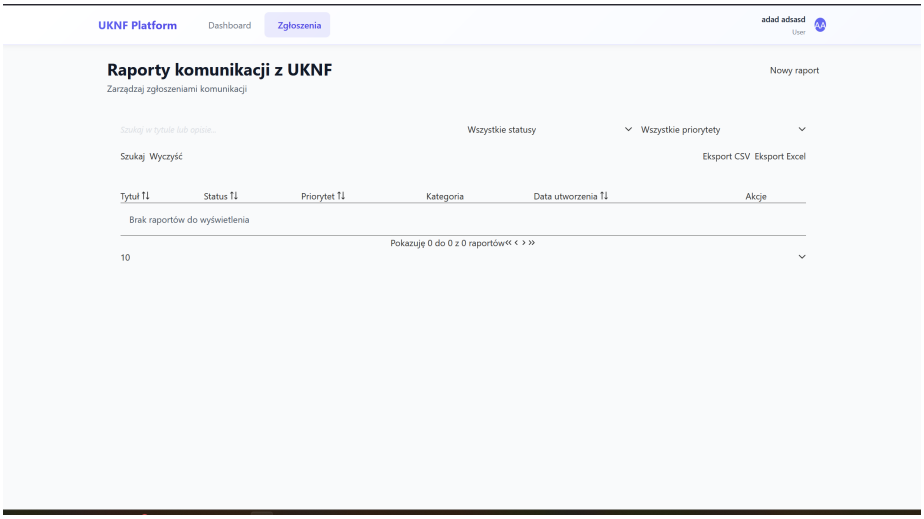

Figure 2: Screenshot 2025-10-05 042104.png

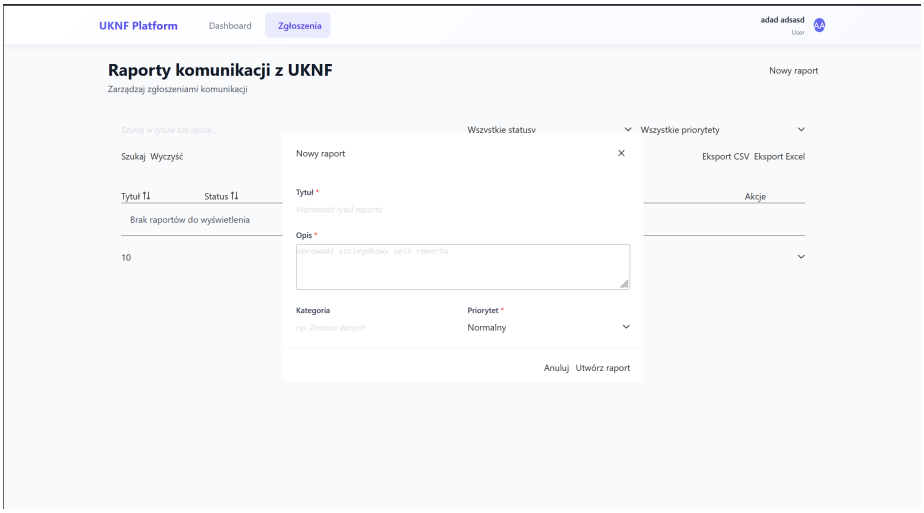

Figure 3: Screenshot 2025-10-05 042111.png

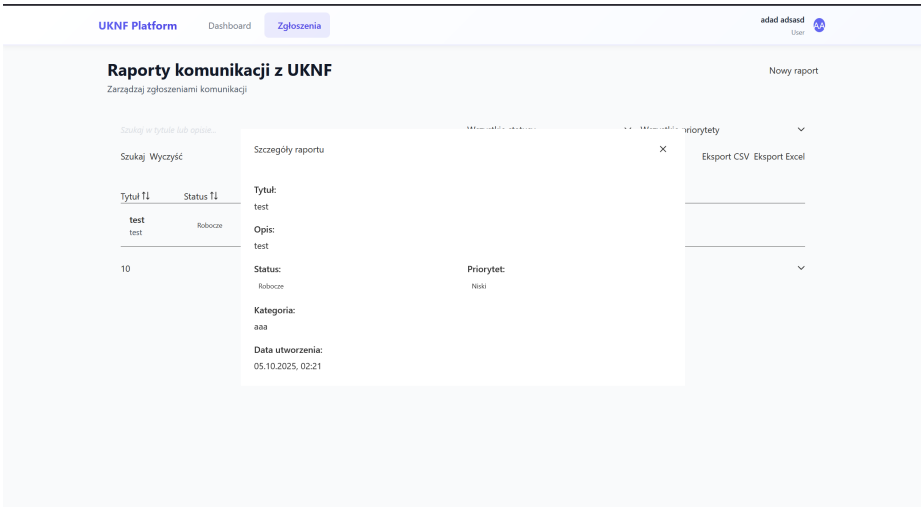

Figure 4: Screenshot 2025-10-05 042134.png

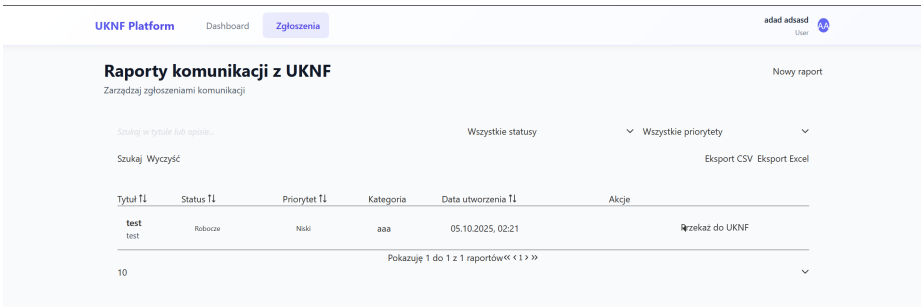

Figure 5: Screenshot 2025-10-05 042144.png