Crossroads

Programming and Test Exercise: Practicing Gen-AI Skills through a Simulation Project

Content

1 Introduction

This individual programming exercise is designed primarily to give engineers a hands-on opportunity to learn and practice using generative AI (gen-ai) tools in the context of software development and test automation. The assigned project—a simulation and visualization of traffic lights at a busy crossroad—serves mainly as a means to facilitate practical gen-ai exploration rather than to deeply study traffic control systems.

Participants will create a software application that models and visually represents traffic light changes. The main objective is to integrate gen-ai throughout the workflow: for assistance with designing algorithms, optimizing code, generating documentation, or producing visual assets. The scenario provides a concrete base for engineers to experiment with and benefit from AI-driven tools during real development tasks.

Deliverables should include a working simulation, a visualization of traffic light states, and a short reflection on the use and impact of gen-ai throughout the project. The ultimate goal is to build confidence and familiarity with AI integration in day-to-day programming and test development.

2 Integrating Generative AI with GitHub Copilot

2.1 Introduction to Gen-AI and GitHub Copilot in VSCode

Generative AI (Gen-AI) is reshaping software engineering by automating code generation, explanation, and review tasks. GitHub Copilot, a prominent Gen-AI tool, integrates directly with Visual Studio Code (VSCode) to augment developer productivity and streamline workflows. By leveraging Copilot, engineers can generate code snippets, resolve issues, and maintain documentation more efficiently, making it a valuable asset in modern software development environments.

This chapter provides a technical, step-by-step guide for software engineers on using GitHub Copilot in VSCode, covering environment setup, code generation, debugging, testing, and documentation practices. The goal is to equip developers with practical techniques for integrating Gen-AI into their day-to-day programming.

The following sections are so organized:

a. Setup and configuration of the working environment
b. Enhance your working environment
c. Application development, debugging and testing
d. Enhance continuous integration workflow
e. Refactoring, Reviewing and Documentation

Every section and/or subsection is annotated with a difficulty level:

- 🟥 Beginner: expected by everyone without a specific role
- 🟨 Intermediate: from developer to lead engineer
- 🟥 Advance: from lead engineer to architect

Setup and configuration of the working environment

2.1.1 Create a repository in a versioning system 🟥

GitHub is a cloud-based platform for hosting Git repositories that enables developers to store, manage, and collaborate on code using tools like pull requests, issue tracking, and project management. Having the project in one of such systems, is today a defector standard for project maintainability. Here are the directives to create a repository on github.

1. **How to create a new repository:**
   - Click on *New* (usually found in the top-right corner or under your profile menu).
   - Fill in the details: *Repository Name*, *Description* (optional), and *Visibility*
   - *Initialize with README*: Check this option to have a README file created automatically.
   - *Add* a .gitignore file to exclude unnecessary files
2. **How to locally clone a repository**
   - Copy the repository URL (HTTPS or SSH).
   - Run the following command in your terminal: git clone <repository-url>

2.1.2 Setting Up VS Code Working Environment 🟥

An IDE (Integrated Development Environment) is a software application that provides developers with a complete set of tools—like a code editor, debugger, and build system—all in one place to make writing and managing code easier and more efficient. Here are the directives to install VS code and Copilot extension:

1. **Install Visual Studio Code:** Download and install the latest version of VSCode from the official website. Ensure your system meets the recommended requirements for optimal performance.
2. **Install GitHub Copilot Extension:** Open VSCode, go to the Extensions view (Ctrl+Shift+X), search for "GitHub Copilot," and click "Install." Sign in with your GitHub account to activate Copilot.

2.2 Enhance your working environment

2.2.1 Customize VS code workflows 🟥

To streamline and automate your entire development workflow, VS Code lets you customize how your application should be built, run, and debugged. To maximize its support to your needs:

1. **Configuring Compile, Debug, and Run workflow in VSCode:** you can use *tasks.json* file to define custom build and run workflow, and the *launch.json* file for debugging configurations. Copilot can provide suggestions for these configuration files based on your project's language and structure. Once set up, Vs Code UI allows you to quickly build, debug, and run your application with a single click.

2.2.2 Make Copilot's answers meet your expectations 🟨

GitHub Copilot can accelerate development by generating code based on natural language comments or partial code snippets. However, without some context and guidelines you will keep iterating with it about the same concepts and workflows repeatedly. To maximize its effectiveness:

1. **Define user prompt files for Copilot:** Write prompts file to define reusable prompts for common development tasks such as generating git branches, source code, performing code reviews, and any task-specific guidelines or reference architecture instructions to ensure consistent execution and code quality. Copilot can help you with suggestions for these prompt files. Additionally, you take inspiration from the repository https://github.com/github/awesome-copilot
2. **Define user instruction files for Copilot:** Instead of manually including context and examples in every chat prompt, write instruction files to define common guidelines and rules that automatically influence how AI generates source code, documentation and handles other development tasks for you. Copilot can help you with suggestions for these prompt files. Additionally, you take inspiration from the repository https://github.com/github/awesome-copilot

2.3 Application development, debugging and unit testing

Before continuing, recall that you cannot expect correct-by-generation code. Copilot is used to speed up coding related tasks and brainstorming where solutions are evaluated. When any solution is provided, you remain the final evaluator and expert, with Copilot supporting you as a colleague at your side.

2.3.1 Generate code with Copilot 🟥

Do not write code but ask Copilot to generate code based on a natural language description and use cases.

1. Ask Copilot to generate your application by proving a clear, intent-driven description of the desired functionalities. Include examples in the description to narrow the answer and decrease the number of interactions with agent.
2. Review Copilot's suggestions and select, edit, or reject them as appropriate for your use case.
3. Use Copilot for boilerplate code, repetitive patterns, or complex algorithm scaffolding, ensuring you maintain control over logic and security.

NOTE:
user instructions can improve the quality of the generated source code tremendously based on quality standards, expected error-handling logics, documentation and many other code-related aspects (Chapter 1.3).

4. Application-Specific Libraries: Use package managers (like npm, pip, conan, or nuget) to install required libraries. Copilot can assist by suggesting dependency lists in manifest files (e.g., package.json, requirements.txt).

2.3.2 Debugging code and resolving issues 🟨

Recall: Generative AI does not mean correct by generation solution. Depending on internal rules, context and your description you have the most likely answer. Thus, the execution of corner cases, and detection of unwanted behavior remains a human task.

1. **Identifying Errors:** When you encounter warnings or runtime errors, Copilot can suggest fixes. Place your cursor near the problematic code or describe the issue in a comment to prompt Copilot for suggestions. Additionally, you can report the actual behavior (such as a wrong output value) and remark Copilot the one you expected to have.
2. **Resolving Warnings:** Copilot can suggest code modifications to resolve common warnings, such as type mismatches, deprecated API usage, or uninitialized variables.
3. **Debugging Techniques:** Use VSCode's built-in debugger in conjunction with Copilot for step-by-step troubleshooting. Copilot can help generate logging statements or identify likely sources of bugs.

2.3.3 Unit testing 🟥

A good regression test is essential as it helps ensure software quality, stability, and long-term maintainability throughout the development process of the application.

1. **Create unit and integration Tests:** Write comments specifying the function to test and the expected behavior. You can ask Copilot to generate unit and component test templates using popular frameworks (e.g., pytest, Jest, xUnit, catch2, gtest). Do not stop on the "happy path" but ask to cover corner cases.
2. **Create Functional Tests:** Write comments specifying the functionality to test and the expected behavior at application level to ensure user-test cases are covered. Copilot can help you generate complex test scenarios mimicking users' interactions.

2.3.4 Test coverage 🟨

A good regression test is thought to be "good" when no cover case is left out. Test coverage is a technic to evaluate the "goodness" of your regression test.

3. **Generate Test Coverage:** With Copilot, you can define a workflow for your regression tests where test coverage metadata is generated to show in a user readable means what code is tested and untested.

4. **Optimizing Test Coverage:** Review the generated coverage report and ask Copilot to generate test scenario to cover uncovered execution paths.

2.4 Module and System Testing 🟨

Where unit testing and integration testing is in the domain of software development, module and system testing is in the domain of test engineering.

1. **Requirements Analysis:** Leverage AI-powered tools to parse specifications, analyze requirements documents, and extract or even generate testable requirements. These tools can flag ambiguities and suggest clarifications, ensuring the foundation for testing is robust and clear.
2. **Test Plan Creation:** Use AI assistants to help draft comprehensive test plans. They can propose test objectives, scope, resource estimates, and schedules based on project context, past plans, and best practices, accelerating the planning phase and reducing omissions.
3. **Tool Selection:** For the selection of tools to be used, you have to consider the test framework, the availability of libraries, the technology to access the application under test, and the programming language(s) to realize the missing pieces. AI agents can assist you with an overview of available options and the suitability of these options for the problem at hand.
4. **Test Case Creation:** AI tools can auto-generate detailed test cases from requirements or user stories, covering both typical flows and edge cases. They also help maintain and update test cases as the system evolves, ensuring ongoing relevance and high coverage.
5. **Keyword Development:** Modern AI assistants can suggest, refactor, or optimize test automation keywords and reusable actions, enhancing maintainability and supporting efficient expansion of automated test suites.
6. **Reporting:** In the end, we want to showcase the results of our efforts using nice looking reports.

2.5 Enhance continuous integration workflow 🟥

Continuous integration is crucial for maintaining quality throughout a project's development and for promptly raising alerts when standards are not met or current implementation can be broken by new code.

1. **Create pipelines for GitLab:** with Copilot you can generate pipelines that run regression tests, send email alerts, and block new code when errors occur.
2. **Schedule periodic checks:** you can generate pipelines periodically executed by GitLab to check code coverage and raise alerts if code/test quality falls below the required threshold.
3. **Create manual or tag-based actions:** with Copilot you can generate action for GitLab, for instance generate an installable package whenever a tag is added to a Git commit.

2.6 Refactoring, review and documentation

Effective development relies on improving code quality, understanding existing logic, and maintaining clear documentation—practices that support long-term project health with Copilot as a helpful partner throughout.

2.6.1 Refactoring Code 🟥

1. **Improving Structure:** Copilot can be used to suggest refactoring options, such as extracting methods, renaming variables, or simplifying complex statements.
2. **Enhancing Readability:** Ask Copilot to rewrite sections for clarification or to conform code to coding standards. Review and test all refactored code to ensure functionality is preserved!
3. **Explore alternatives:** Use Copilot to suggest alternative implementations or highlight potential issues, but always conduct a thorough manual review for logic, security, and maintainability.

2.6.2 Reviewing 🟨

Code review is essential for quality assurance.

1. **Check for uninitialized variables and race condition**: Ask Copilot to analyze the new code and identify any critical aspect.
2. **Check for coding conventions:** Ask Copilot to analyze the new code and identify any missed code convention and/or violation compared to your reference architecture.
3. **Clarify logic and intent** by writing a comment like #Explain this function above a code block, Copilot can generate summaries or inline comments to clarify logic and intent. Always review these explanations for technical accuracy and completeness before sharing with others.

2.6.3 Documentation and explaining code 🟥

Documentation tends to diverge from source code when it is not updated and maintained. You can use Copilot to update and check if source code follows the intent described.

1. **Generate summaries** of code by prompting Copilot with comments like #Summarize this function's changes.
2. **Generating Documentation:** Leverage Copilot to create or update README.md files, API docs, and inline comments. Prompt detailed descriptions to receive relevant sections or summaries.
3. **Create user prompt file** to automatically detect technology stacks and architectural patterns, generate visual diagrams, documents implementation patterns, and provides extensible blueprints for maintaining architectural consistency and guiding new development.

2.7 Conclusion

Integrating GitHub Copilot with VSCode empowers software engineers to automate repetitive tasks, enhance productivity, and maintain high code quality. By following these technical guidelines, developers can effectively leverage Gen-AI throughout the software development lifecycle—from initial setup and code generation to testing, explanation, and documentation. Continuous practice with Copilot will deepen familiarity with AI-driven workflows and support ongoing professional growth in an evolving field.

3 Assignment: Traffic Lights at a Crossroad

3.1 First Iteration: Basic application

The basic application for this assignment consists of three main components:

- **Simple Traffic Generator:** This module is responsible for simulating the flow of vehicles approaching and entering a crossroad. It creates virtual traffic patterns, such as cars arriving at random intervals and from different directions, to mimic real-world conditions at an intersection.
- **Basic Traffic Light Control Algorithm:** The application includes a straightforward logic for managing the traffic lights at the crossroad. This algorithm determines when each light turns green or red, coordinating the movement of vehicles through the intersection to ensure safety. This control algorithm is purely clock driven and does not consider any traffic information.
- **Results Visualization on a Map:** The outcome of the simulation—such as vehicle movement, stops, and light changes—is displayed on a map within the application interface. This visualization allows users to observe how traffic flows and how the algorithm affects congestion and wait times at the crossroad.

This foundational setup provides a practical environment to experiment with and improve traffic control logic and visualize the impact of different strategies.

3.2 Iteration: Crossroads viewer and editor

To ensure crossroad layouts are reusable and accessible, each configuration must be saved, for instance stored in a database. This approach not only provides flexibility for experimentation and deployment but also establishes a structured way to manage multiple intersection designs within the application.

3.3 Iteration: Safety design

Before we start working on smarter control algorithms, we need to ensure that the crossroads are always in a safe state. Therefore, in between the control algorithm and the interface to the traffic lights, a safety checker will be added. The safety checker warrants the consistency of commands to the traffic lights. The safety checker implements a series of rules and rigorously checks that none of the rules are violated at any time.

This approach ensures that conflicting signals—such as multiple directions showing green at the same time—are systematically prevented. The safety checker acts as a safeguard layer, monitoring all light changes and enforcing critical safety constraints regardless of the logic produced by the control algorithm. By introducing this component, the integrity and reliability of the intersection control system are significantly strengthened, reducing the risk of accidents caused by software errors or unforeseen circumstances.

3.4 Iteration: Different Control Algorithms

To enable smarter traffic management, the system needs to support multiple control algorithms and provide a mechanism for switching between them seamlessly. This begins with designing the application architecture so that different algorithms—such as the basic clock-driven control, the “null-control” (amber flashing mode), and more advanced logic—can coexist and be selected as needed. The ability to switch algorithms dynamically is crucial for both operational flexibility and safety.

Implementing a “null-control” mode, which causes all lights to flash amber, serves as a vital fallback. This mode should be triggered automatically if the safety checker detects any rule violation or inconsistency, ensuring the intersection never enters an unsafe or ambiguous state. When switching between algorithms, it is essential to introduce transitional logic that places all lights in a safe, well-defined state (such as all red or all flashing amber) before activating the next algorithm. This transitional period prevents conflicting signals and avoids illegal states, such as multiple greens in different directions.

Once robust algorithm management and safe switching are established, the system can evolve to include smarter algorithms that utilize input from traffic counters placed upstream in the lanes. These counters provide real-time data on approaching vehicles, enabling the control logic to optimize green light durations and reduce congestion proactively. By layering these enhancements, the intersection control system becomes both safer and more responsive to actual traffic conditions.

3.5 Iteration: Key Performance Indicators

A simulation scenario includes three main components:

- The intersection itself, featuring its traffic lights and sensors
- The algorithm that manages control
- Configuration options for the traffic generators

To assess the effectiveness of the control algorithm within a given scenario, it is necessary to establish measurable metrics, known as Key Performance Indicators (KPIs). These KPIs provide a quantifiable means of expressing the quality of control and enable objective evaluation of different algorithms and intersection configurations.

3.6 Iteration: Recording and Playback

The ability to record and replay scenarios is needed. This functionality allows for the preservation of specific simulation runs, facilitating thorough analysis and comparison of results. Playback capabilities ensure that scenarios can be revisited, helping refine algorithms and optimize intersection performance based on recorded data.

3.7 Iteration: ORM (only with a relational database)

The application becomes more and more dependent on its database. Traditional methods, accessing the database directly using query language statements such as SQL, ignore the fact that there is major commonality between the data structure in the database and the object model in the application. Enter Object Relation Mapping (ORM).

An Object-Relational Mapping (ORM) is a programming technique that creates a bridge between object-oriented programming languages and relational databases. An ORM is a layer of abstraction that converts data between incompatible type systems - specifically between your application's objects and your database's tables, rows, and columns.

Key Benefits:

- Abstraction of SQL: Write database operations using your programming language instead of SQL

- Object-Oriented Interface: Interact with your database using familiar object-oriented patterns
- Reduced Boilerplate: Eliminate repetitive CRUD (Create, Read, Update, Delete) code
- Database Agnosticism: Switch database systems with minimal code changes
- Data Validation: Built-in validation before persisting data

ORMs provide a convenient way to work with databases while maintaining the object-oriented paradigm in your application code.

In this iteration, the app is to be refactored to use an ORM instead of direct access to the database.

3.8 Iteration: Crossroads viewer and editor

To support applying the program to various road crossings, the application should incorporate a crossroads editor that enables users to design, modify, and select different intersection layouts. This editor will allow for the creation and customization of road crossing maps, which can be tailored to reflect real-world or hypothetical scenarios.

3.9 Iteration: Decoupling of UI

Up till now, the UI has been an integral part of the application. Moving forward, we want to split the UI from the simulator and achieve the following:

- We want to close and restart the UI while the simulation keeps running. Ideally, we want to alter the UI without losing the state of the simulator.
- We want to instantiate multiple UIs, connected to the same simulation.
- We want to run the UI on a different computer than the simulation.

As usual, take care of security when connecting and running applications via a network.

3.10 Iteration: Live Maps

To enhance the user experience, it is beneficial to load actual map data into the UI. Maps can be used from OpenSteetMaps or Google Maps.

This iteration is also suited to review and refine the traffic generators to make scenarios more realistic. E.g. trucks have different characteristics than passenger cars and thus behave differently in traffic.

Editor or contact
Capgemini
Limburglaan 24
Eindhoven

Author
Paul van Haren
Solution Architect

Publication year
January 2026