Supplementary Answer of Q3 of Reviewer A

To illustrate how APICopilot Deals with Runtime-Dependent Arguments, we take the following example

```java
// Represents a configuration provider
class Config {
    private int configuredPort = 8080; // Default
    private int configuredTimeout = 5000; // Default

    public Config() {
        // Attempt to load from environment variables at runtime
        String envPort = System.getenv("MY_APP_PORT");
        if (envPort != null) {
            try {
                this.configuredPort = Integer.parseInt(envPort);
            } catch (NumberFormatException e) {
                System.err.println("Warning: Invalid MY_APP_PORT. Using default.");
            }
        }
        String envTimeout = System.getenv("MY_APP_TIMEOUT_MS");
        if (envTimeout != null) {
            try {
                this.configuredTimeout = Integer.parseInt(envTimeout);
            } catch (NumberFormatException e) {
                System.err.println("Warning: Invalid MY_APP_TIMEOUT_MS. Using default.")
    ;
            }
        }
    }

    public String getPort() { // Returns port as String
        return String.valueOf(this.configuredPort);
    }

    public int getTimeoutMilliseconds() {
        return this.configuredTimeout;
    }
}
class Server {
    public void start(String port, int timeoutMilliseconds) {
        System.out.println("Server: Initializing on port: " + port +
                           " with timeout: " + timeoutMilliseconds + "ms.");
        // ... server startup logic ...
        try {
            // Simulate server startup work
            Thread.sleep(500);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("Server startup interrupted.");
        }
        System.out.println("Server: Successfully started on port " + port);
    }
}
public class MainApplication {
    public static void main(String[] args) {
        Config applicationConfig = new Config();
        Server myWebServer = new Server();

        String serverPort = applicationConfig.getPort();
        int connectionTimeout = applicationConfig.getTimeoutMilliseconds();

        // Developer types: myWebServer.start(  <-- APICopilot needs to suggest
    arguments here
        // The goal is for APICopilot to suggest 'serverPort' and 'connectionTimeout'.

        // Line where APICopilot would provide suggestions
        myWebServer.start(serverPort, connectionTimeout); // Completed call

        System.out.println("\nMainApplication: Setup complete.");
    }
}
```

Listing 1: Config.java

APICopilot handles these runtime-dependent arguments effectively not by trying to predict their specific runtime *values*, but by:

1. Symbolically understanding the variables and their origins within the current code.
2. Leveraging knowledge of common coding patterns from a vast corpus of examples.
3. Using this combined information to guide a Large Language Model (LLM) in making appropriate suggestions.

Here's a step-by-step breakdown using the example above, when the developer types `myWebServer.start(`:

Dynamic Contextual Understanding

As the developer writes the code in `MainApplication.java`, specifically:

```
String serverPort = applicationConfig.getPort();
int connectionTimeout = applicationConfig.getTimeoutMilliseconds();
```

APICopilot processes this preceding code to build an input Knowledge Graph (G_{input}). This graph doesn't store the runtime value of `serverPort` (e.g., "8080") because that's not known yet. Instead, it symbolically captures:

- A variable named `serverPort` of type `String` exists.
- `serverPort` is derived from the method call `applicationConfig.getPort()`.
- A variable named `connectionTimeout` of type `int` exists.
- `connectionTimeout` is derived from `applicationConfig.getTimeoutMilliseconds()`.

This symbolic representation is crucial because it tells APICopilot what variables are available and how they were formed, regardless of their future runtime values.

Learning from Similar Usages

APICopilot has access to a pre-processed knowledge base of code examples (KG_{example}). When the developer attempts to call `myWebServer.start(`, APICopilot searches this knowledge base for contextually similar code snippets. It might find numerous examples where:

- Variables initialized from configuration objects are passed to server initialization methods.
- Methods like `config.getPort()` or `settings.getServerPort()` provide arguments for parameters named `port`.
- Methods like `config.getTimeout()` provide arguments for parameters related to `timeout`.

These examples reveal established patterns of how developers typically handle such runtime-associated variables in API calls.

Informed LLM Prompting

With the symbolic information from G_{input} (knowing `serverPort` and `connectionTimeout` are available and how they were obtained) and the usage patterns from KG_{example}, APICopilot constructs a highly informative prompt for an LLM. This prompt might conceptually include:

- "The user is calling `myWebServer.start(String port, int timeoutMilliseconds)`."
- "Available in the current scope are:
  - `String serverPort` (obtained from `applicationConfig.getPort()`).
  - `int connectionTimeout` (obtained from `applicationConfig.getTimeoutMilliseconds()`)."
- "Similar code examples frequently use variables derived from configuration port methods as the 'port' argument and variables from configuration timeout methods as the 'timeout' argument."

Recommendation

The LLM, guided by this rich, contextual prompt that highlights both the available symbolic variables and relevant usage patterns, can then accurately infer and recommend:

- The variable `serverPort` for the `port` parameter.
- The variable `connectionTimeout` for the `timeoutMilliseconds` parameter.

Thus, APICopilot successfully suggests the correct runtime-dependent variables by understanding their role and origin within the code's structure and by recognizing common programming patterns, all without needing to know their specific values at the moment of suggestion.