Code Review and Automated Testing

Reviewing Code

A) DroneServiceImpl.java

FILE HEADER: Are the following items included and consistent?	***yes***	***no***	***comments***
Author and current maintainer identity		✓	No author and maintainer information included. (The repository was created for coursework so was not needed but might be helpful in production).
Overview of class purpose like is the class the principal entry point	✓		Javadoc explains the role of the service.
IMPORT SECTION: Are the following requirements satisfied?	***yes***	***no***	***comments***
Each import relevant and used		✓	Unused import *(@Autowired*) - dependency injection is done via the constructor, so this import should be removed.
CLASS DECLARATION: Are the following requirements satisfied?	***yes***	***no***	***comments***
The constructor is explicit if the class (if the class is not static)	✓		An explicit constructor exists and uses dependency injection for the required attributes.
Visibility and structure are appropriate	✓		A public service implementation is appropriate, as it is accessed by the controller, which is independent of the service class.
CLASS DECLARATION JAVADOC: Does the Javadoc header include:	***yes***	***no***	***comments***
One sentence summary of class functionality	✓		This is provided.
Usage instructions		✓	The purpose of the class is described, but there are no explicit usage notes (e.g., how the service is injected and used by the controller).
CLASS: Are names compliant with following rules?	***yes***	***no***	***comments***
Class: CapitalizedWithEachInternalWordCapitalized Special Case: If class and interface have same base name, distinguish as ClassName and ClassNameImpl	✓		*DroneServiceImpl* is correct and distinguishable from the *DroneService* interface.

Constants use ALL_CAPS_WITH_UNDERSCORES		✓	stepSize is inconsistent with this requirement.
Field name: capsAfterFirstWord, name must be meaningful outside of context	✓		This has been implemented.
IDIOMATIC METHODS: Are names compliant with the following rules?	***yes***	***no***	***comments***
MethodName: capsAfterFirstWord, meaningful Local variables:capsAfterFirstWord		✓	*getnextPoint* should be *getNextPoint* and *positionInRegionCheckForAStar* is unclear, and returns true when unsafe → rename for correctness and readability. Local variables consistent with the requirement.
EXCEPTIONS: Are all exceptions handled?	***yes***	***no***	***comments***
Exceptions are handled or surfaced appropriately	✓		*hasMatchingAttribute* and *hasQueryAttributes* silently return false on exceptions, which is appropriate given the specification that non-existent drone attributes in the input should be ignored.
FORMATTING: Is the code properly formatted?	***yes***	***no***	***comments***
Formatting is consistent and readable	✓		Code is readable overall – class has several methods but those interacting with each other grouped together so easier to navigate.
PERFORMANCE: Are avoidable inefficiencies present?	***yes***	***no***	***comments***
Repeated expensive calls are avoided		✓	*getDroneDetails* class called repeatedly due to the for loop within *getAvailableDronesLogic*
Shared mutable states avoided	✓		Local maps introduced in *getAvailableDronesLogic* prevent concurrency issues, ensuring successful response on first try.

Based on the above review, the following to-do list was identified (in priority order):

- Avoid calling /drones repeatedly via *getDroneDetails()* inside loops (especially in *getAvailableDronesLogic*).
- Rename *positionInRegionCheckForAStar* to reflect behaviour (since true indicates "unsafe/violates restricted area")
- Rename *stepSize* to a constant-style name (e.g., STEP_SIZE).
- Rename *getnextPoint* to *getNextPoint*.

- Remove import *org.springframework.beans.factory.annotation.Autowired* as it is not being used.
- Add a brief note in Javadoc stating the service is injected into the controller via dependency injection (just 1–2 lines).
- (Optional) Add an author/maintainer note to the class header for production maintainability.

B) ServiceController.java

FILE HEADER: Are the following items included and consistent?	***yes***	***no***	***comments***
Author and current maintainer identity		✓	No author and maintainer information included. (The repository was created for coursework so was not needed but might be helpful in production).
Overview of class purpose like is the class the principal entry point	✓		Javadoc explains the role of the service.
IMPORT SECTION: Are the following requirements satisfied?	***yes***	***no***	***comments***
Each import relevant and used	✓		Imports align with Spring MVC + DTO usage.
CLASS DECLARATION: Are the following requirements satisfied?	***yes***	***no***	***comments***
The constructor is explicit if the class (if the class is not static)	✓		An explicit constructor exists and uses dependency injection for the required attributes.
Visibility and structure are appropriate	✓		A public controller is appropriate, as it acts as the entry point for HTTP requests and delegates functionality to the service layer.
Class responsibility is clear (controller layer only)	✓		Controller mostly delegates to service layer; minimal logic is present.
CLASS DECLARATION JAVADOC: Does the Javadoc header include:	***yes***	***no***	***comments***
One sentence summary of class functionality	✓		A clear one-line summary is included. However, the remaining Javadoc closely mirrors the service class description, so it could be made more controller-specific.
Usage instructions		✓	The endpoints and functionalities are described, but there are no explicit usage notes (e.g., explaining that the controller

			maps HTTP requests to service-layer operations).
CLASS: Are names compliant with following rules?	***yes***	***no***	***comments***
Class: CapitalizedWithEachInternalWordCapitalized	✓		*ServiceController* is correct
Field name: capsAfterFirstWord, name must be meaningful outside of context	✓		This has been implemented.
IDIOMATIC METHODS: Are names compliant with the following rules?	***yes***	***no***	***comments***
MethodName: capsAfterFirstWord, meaningful Local variables:capsAfterFirstWord		✓	Method and local variable naming is consistent, and controller method names mirror the service operations they call.
FORMATTING: Is the code properly formatted?	***yes***	***no***	***comments***
Formatting is consistent and readable	✓		Code formatting is consistent and easy to follow. Endpoint methods are short and structured similarly, mainly returning *ResponseEntity* objects and delegating logic to the service layer.
PERFORMANCE: Are avoidable inefficiencies present?	***yes***	***no***	***comments***
Repeated expensive calls are avoided		✓	*getDroneDetails* class /drones repeatedly due to the for loop within *getAvailableDronesLogic*
Shared mutable states avoided	✓		Local maps introduced in *getAvailableDronesLogic* prevent concurrency issues, ensuring successful response on first try.
ENDPOINTS: Are the following requirements satisfied?	***yes***	***no***	***comments***
Endpoints return appropriate HTTP status codes	✓		404 is returned for missing drone/medicine and 400 for invalid region/angle. Successful requests return 200 OK.
Validation is applied to request inputs		✓	@Valid is used for most request bodies, but it is missing on some endpoints (e.g., *placeOrder* and *showFlightPath*), so input validation is not fully consistent.
Error responses are consistent and informative		✓	Error responses are consistent in status codes, but not very informative because many return body(null) rather than a structured error message.
INPUT VALIDATION: Are edge cases handled safely?	***yes***	***no***	***comments***

Inputs validated appropriately	✓		Bean validation is used for most inputs. Additional constraints that cannot be expressed through annotations (e.g., region closure and angle multiple-of-22.5) are checked manually in the controller.

Based on the above review, the following to-do list was identified (in priority order):

- Apply consistent request validation by adding @Valid to endpoints where it is currently missing (e.g., *placeOrder* and *showFlightPath*), ensuring input validation is consistent across the API.
- Improve error response informativeness by returning a simple structured error message instead of body(null) for 400 and 404 responses (while keeping responses minimal for security).
- Make the class Javadoc more controller-specific by describing its role as the midpoint between the service and the user interface that maps HTTP requests to service-layer operations.
- Add brief usage notes in the Javadoc explaining how the endpoints are accessed (e.g., via /api/v1/…) and that this controller delegates work to the service layer.
- (Optional) Add an author/maintainer note to the class header for production maintainability.

CI Pipeline

The CI Pipeline was constructed using GitHub Actions and is triggered on every push and pull request to the main branch. The pipeline contains three jobs: build, test, and lint. If one of these jobs fail, then the changes in code are not pushed to the working branch. The pipeline performs the following tasks:

- Checks out the repository code (*actions/checkout@v4*)
- Sets up a consistent Java environment using *Temurin JDK 21*
- Uses Maven dependency caching to improve build efficiency (*cache: maven*)
- Runs the unit and integration tests using Maven (*mvn -B test*)
- Uploads test reports from *target/surefire-reports/* as an artifact (*actions/upload-artifact@v4*) even if tests fail
- Packages the application into a runnable artifact (*mvn -DskipTests package*)
- Runs Checkstyle linting (*mvn -B checkstyle:check*) in the lint job, enforcing Google's Java style rules by checking for formatting and consistency issues; any failures will cause the workflow to fail.

Note that the Junit unit-level and integration-level tests implemented between *LO2* and *LO3* are automatically run due to the test job and code style compliance is enforced via the lint job. Hence, testing and quality checks are embedded into the CI pipeline. This ensures

- verification of the core correctness properties,
- a consistent test environment across machines and
- immediate feedback to detect failures triggered by changes through the Surefire test reports.

Based on the above discussion, some of the expected pipeline functions include:

1) The workflow stops and the build job does not run if any tests fail, since the build stage depends on the test stage (needs: test).
2) Evidence of the executed tests and their outcomes is available through the uploaded test reports (*target/surefire-reports/*), which can be inspected even if failures occur.
3) If *Checkstyle* fails, the lint job fails and the workflow fails (quality gate enforced).
4) Identification of common errors**,** such as:
    a) Compilation errors, such as missing imports or broken method calls, which would cause the workflow to fail during the build/package stage.
    b) Functional requirement regressions**,** where changes that break route validation logic (or other core functionality) would be detected through failing JUnit tests in the test stage.

References

Tutorial Inspiring CI Pipeline