npj | artificial intelligence

Article

https://doi.org/10.1038/s44387-026-00101-6

AI generated drone command and control station hosted in the sky

Peter J. Burke ✉

Advances in artificial intelligence (AI) including large language models (LLMs) and hybrid reasoning models present an opportunity to reimagine how autonomous robots such as drones are designed, developed, and validated. Here, we demonstrate a fully AI-generated drone control system: with minimal human input, an artificial intelligence (AI) model authored all the code for a real-time, self-hosted drone command and control platform, which was deployed and demonstrated on a real drone in flight. We quantitatively benchmark system performance, code complexity, and development speed against prior, human-coded architectures, finding that AI-generated code can deliver functionally complete command-and-control stacks at orders-of-magnitude faster development cycles, though with identifiable current limitations related to specific model context window and reasoning depth. Our analysis uncovers the practical boundaries of AI-driven robot control code generation at current model scales. Not a single line of code was written by a human. A machine built a robot's brain.

In Arnold Schwarzenegger's Terminator, the robots become self-aware and take over the world. While the outcome needs to be prevented, the concept caught the imagination of the world, and suggested a possible avenue for AI applications in robotics: AI controlled design, manufacture, and control of robots and drones: Robots building and designing other robots. In this paper, we take a step in that direction: An AI code writing machine creates, from scratch, with minimal human input, the control station of a drone.

Just like the human brain is divided into a hierarchy of functions from the most basic to the most advanced, drone control software contains low, intermediate, and high level control systems. The low-level firmware ("flight controller") such as Ardupilot[1], Pixhawk PX4[2] Betaflight[3], etc. runs at the base layer and sends commands directly to the motors, in order to keep the drone level without pilot input, for example. They interpret pilot commands from the position of fingers and joysticks on a remote control. Ardupilot has a million lines of code which is beyond the scope of AI to create from scratch as of this writing (summer 2025). In any case, such a low-level code requires extensive development and is not the scope of this work.

A higher level computer, with on-board image processing and higher-level decision making is typically Robot Operating System (ROS[4]) or some other autonomous code with general goals such as collision avoidance, but not pilot controlled. ROS2 also has around 2.5 million lines of code. Creating an AI written version of ROS2 or equivalent is also probably not possible with current AI models, just like writing Linux from scratch is also not possible with current AI models. Rather, the scope of this work is the intermediate code, typically called the "ground-control-station", which we discuss next.

A "Ground control station (GCS)" is a command and control station that sends commands to a drone and receives telemetry back from the drone. It presents data in a uniform manner such as the drone position on a map, and allows higher-level commands to be sent to the drone, such as waypoint missions, auto- land, auto-take off, which is one level higher than the base firmware. It is referred to as a GCS, since it is a computer program on the ground that controls the drone in the air through a wireless telemetry link. Mission Planner[5] and QGroundControl[6] are examples. They require a dedicated Linux, Mac, or Windows machine and use the windowing system of the operating system (OS).

In Fig. 1, we show the variety of drone wireless links in use today. In the standard "stick and rudder" method, a pilot uses a 4 channel radio (aileron elevator throttle rudder AETR) to manually pilot the drone, with an on-board radio receiver (RX) on the drone. In order to provide some automated computer control of the drone, a digital wireless link sending commands to the drone is used from a computer on the ground running GCS software, directly to the drone flight controller (FC) in the air, through some sort of modem. This can be a WiFi transceiver (Fig. 1B), a second bit stream together with manual control (ExpressLRS MAVLink, Fig. 1C), or a dedicated RF modem such as the RFD900[7] (Fig. 1D). The common command set used by drones using the Ardupilot firmware is MAVLink[8], a set of low- and high-level commands for drone command, control, communication, and telemetry. Normally, the GCS is a standalone executable installed and running on a Linux, Mac, or Windows PC. The drone pilot usually is physically collocated with the PC, as the OS GUI is needed for the GCS to function properly.

Department of Electrical Engineering and Computer Science, University of California, Irvine, CA, USA. ✉e-mail: pburke@uci.edu

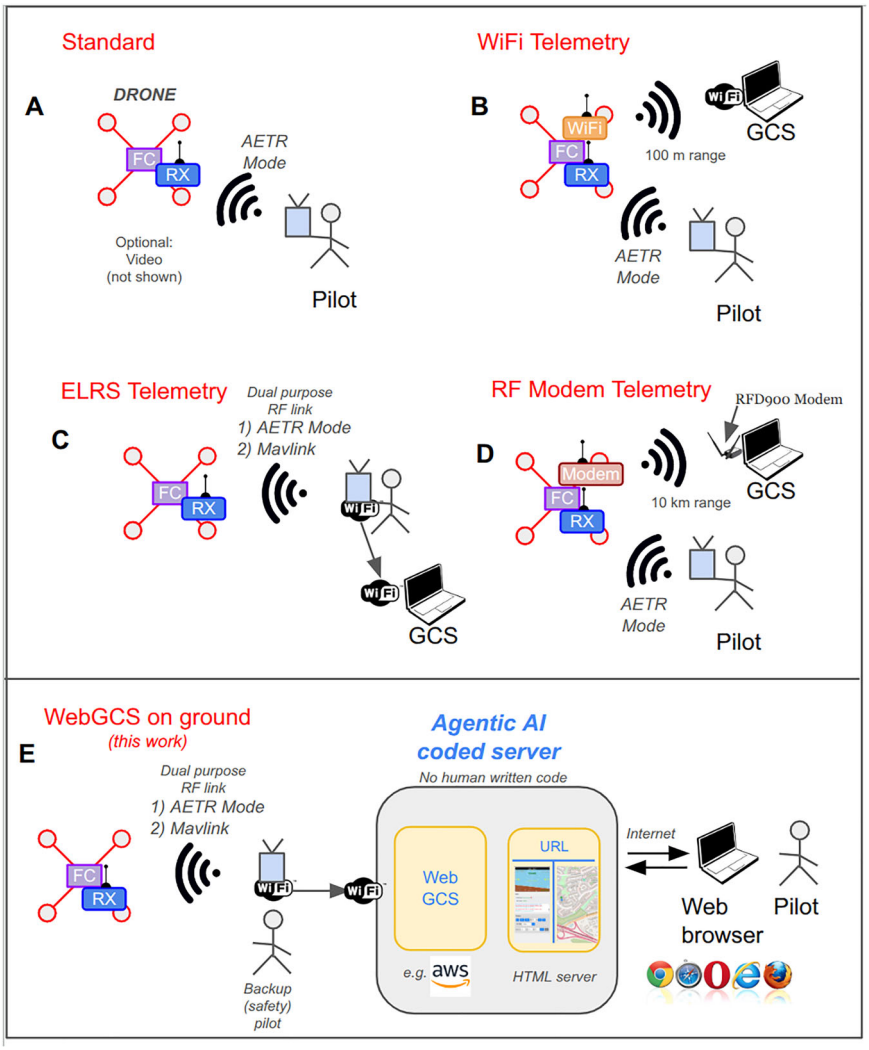

Fig. 1 | Ground control stations. A Standard setup: no GCS, only manual pilot control of aileron, elevator, rudder, throttle (AETR): "Stick and rudder". **B** WiFi telemetry **C** ELRS telemetry **D** RF Modem Telemetry **E** Web ground control station coded entirely by AI (this work.) FC flight controller, RX receiver. See text for detailed explanation.

A "web" GCS replaces the standalone executable and replaces it with a web based equivalent set of functionalities. The advantage is the pilot does not need to be collocated near the PC, enabling piloting the drone from anywhere in the world: The control station is also a website host, accessible from any browser, meaning it can be run remotely from anywhere in the world, on any OS, including tablets and smartphones. To be clear, this is *not* just a website, it includes a full stack GCS. The website is just the interface to the backend. This means any pilot that can access the website hosted on the ground computer can remotely pilot the drone, a significant extension over existing GCS programs such as Mission Planner and QGroundControl, which require access to the terminal of the computer running the custom GCS code. This adds additional flexibility of different operation systems for the pilot, and different locations.

We first invented this in 2020 with extensive, manual coding (years of effort)[9,10]. In fact, some entire companies (reviewed by us in refs. 9,10) are based on selling proprietary versions, so it is not unreasonable to estimate that millions of dollars in human time (developers) have gone into web based GCS development globally. In our experience, we have had three generations of undergraduate and graduate students develop increasingly sophisticated web-based GCSs. One version of these was used to set a Guinness aviation record for the longest distance to pilot a drone (around the world)[11], demonstrating the power of cloud based, web enabled command and control of drones.

All of the above was coded manually. To our knowledge (see "prior art" literature review section below), generative AI has never been used for coding any levels of the drone "brain", i.e., firmware (low level brain) (Ardupilot, PX4), GCS (Mission Planner, QGroundControl) (intermediate level brain), or high level control (ROS2) (high level brain). All of that code represents millions of lines of code and easily hundreds of person-years of effort in manual coding. Writ large, this is a huge opportunity for generative AI to contribute to drone coding.

In order to begin this journey, in this paper, we develop and demonstrate a process for generative AI to code a web based GCS (RESULT 1 of 2). In this work, AI is used to code a GCS on the ground shown in Fig. 1E. The entire intermediate software layer (the control station) was coded in about 2 weeks of human time. The development process, function, testing, simulation, and flight demonstration is presented in this paper. We name the AI coded GCS "WebGCS". We anticipate that this is the beginning of a paradigm shift in how drone code is written, tested, and deployed, and as such is an important "first" milestone in our (long-term) vision: Robots building robots brains. The fact that we developed a web based GCS adds the possibility to host the GCS in the cloud. (For example, with ELRS MAVLink the radio on the ground provides a WiFi connection, it can be connected to the internet.) Although to date all web GCS projects have been hosted "on the ground", the code can actually run on a computer on the drone (called a companion computer), making the drone a flying website. To our knowledge, creating a website in the sky has not yet been demonstrated. RESULT 2 of 2 in this paper is the deployment of the WebGCS on a computer on the drone itself (Fig. 2). This is a robot control station in the sky, coded by a machine.

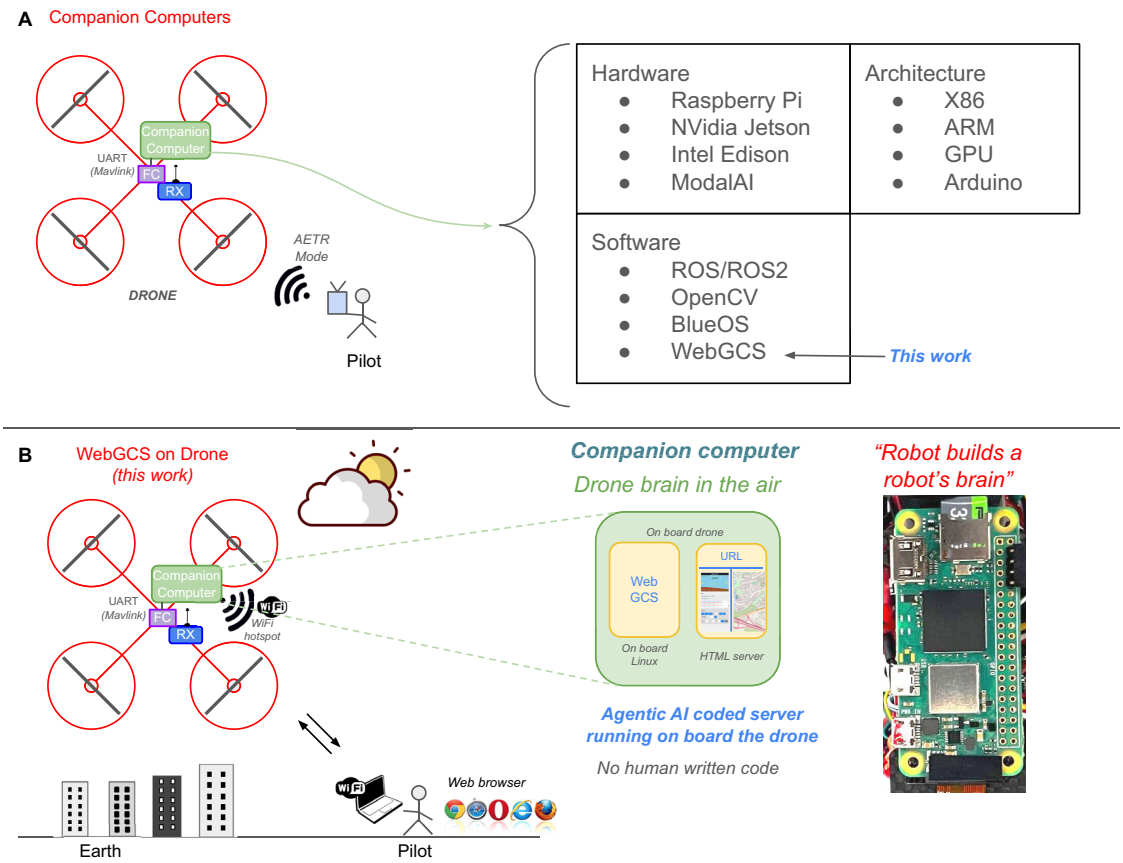

Fig. 2 | On board computers. A State of the art of companion computers (hardware/software) before this work. All software was written line by line by *humans*. **B** A flying web based drone control system (this work), written entirely by *machines*.

Results

Result #1 of 2: "The Process" (Generative AI for drone code)

The "process" we demonstrated in this paper is actually part of the results.

We next describe the *Pipeline*. The pipeline includes coding, deployment, and flight testing (real or simulated), and is shown in Fig. 3. Once the code is sufficiently developed by the AI, it is then deployed to either a ground based control station (GCS), or a control station on the drone itself. When deployed on the ground, the code can run on a local machine or in the cloud, as long as an internet connection from the drone to the GCS is maintained. The pilot then uses a web browser to log into the WebGCS to control the drone. When deployed on the drone in the air, the drone hosts the website directly from the air. The pilot can connect (log into) the drone from a local WiFi connected PC as the drone actually creates a WiFi hotspot in the air. In addition, if desired, the TCP traffic can be routed to any computer in the world, where the pilot can connect and control the drone from.

We next describe the *Process: Methods used to build the software (prompting strategy, which models were used etc.)*. The overall methods used to build the software progressed through three phases. In phase I, only chat in windows with cut/paste was used. Once it was proved AI was a viable approach, we moved to Phase II where an IDE was used. This worked for a while but required a lot of manual curation. Phase III used automated testing to remove the human from the loop. We discuss these and the models, dates, and sprints used in these phases. See Supplementary Information for dates, models, tokens, hours, lines of code, commits, milestones, etc.

All of these sprints and phases (starting with Phase II) eventually used GitHub commits to prevent deletion of old, functional code by the AI, and to track progress. Table 1 shows the GitHub LOCs for the sprints.

During this period (spring/summer 2025), it was not common practice for the model providers to save the prompts to memory. Only some of our prompts were manually recorded, and they were deleted by the providing industry. Therefore, it is not possible to have the texts for each sprint more standardized. We are pleased that during the writing of the revision of this paper (January 2026), most major providers have begun keeping track of prompts. We are pleased that industry is following our suggestion from the original preprint version of this paper (dated Aug. 4, 2025), "Request for industry" (still in this version, Supplementary Information) to develop an industry standard for data and record retention, including prompts, hours, tokens, etc. For example, 3 months after our preprint, Anthropic did take our suggestion and enable memory[12]. At the moment, prompts are saved, but hours and tokens are still not transparently tracked by industry. Thus there is as of yet no industry standard for tracking AI code development.

Phase I: Chat windows cut paste (Sprints 1,2) In this phase, we began with an elemetary AI task: write a Python script to make a drone takeoff or land. This could be done with a simple chat window in a webbrowser. The process was to cut and paste the Python script into a text file, then manually run it. Eventually a full install script (single file) to install all of the files needed onto a fresh Linux instance, including the HTML files, Javascript, Linux Systemd service files, Python code, environment setup, etc. was created by the AI. Sprint 1 ran out of tokens before a flight worthy code was created. Sprint 2 used a larger token model which enabled a functional prototype to be demonstrated in a real flight. The context limit is very important, and will be discussed in detail below. We now describe sprints 1,2 in detail.

Sprint 1: Claude in browser The first sprint used Claude[13] in a browser (where the code would be copied/pasted from the browser when ready), with some very simple prompts (actual prompts):

- **Prompt:** Write a Python program to send MAVLink commands to a flight controller on a Raspberry Pi. Tell the drone to take off and hover at 50 feet.

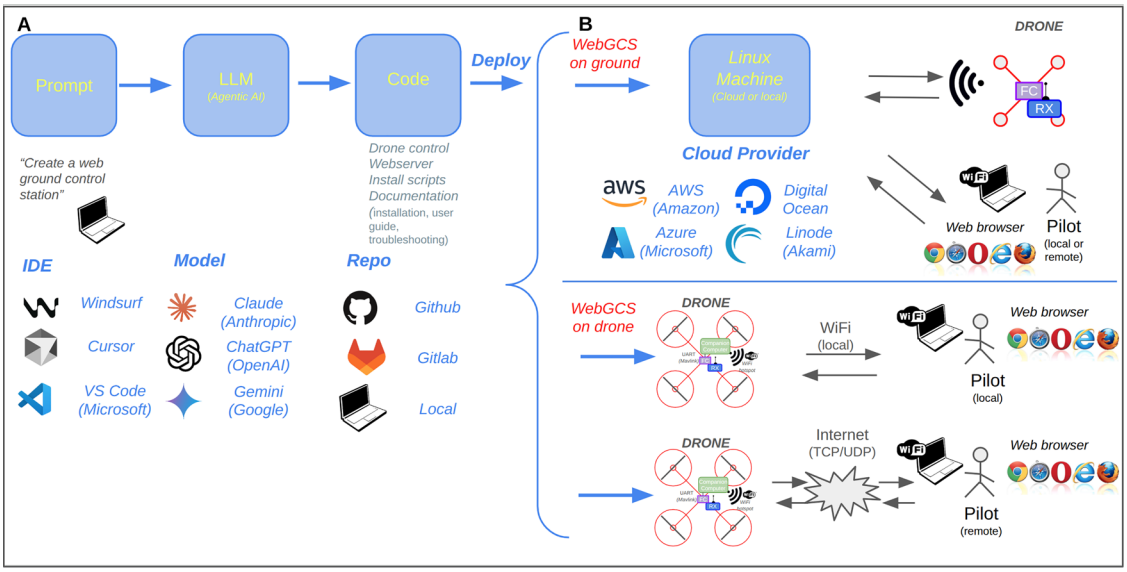

Fig. 3 | Pipeline. A Development pipeline. **B** Deployment pipeline.

Table 1 | Code changes across development sprints showing additions and deletions by date

DateTime	Additions	Deletions	Sprint
4/20/25	3,847,658	−15	II
4/27/25	1,923,210	−5,767,157	
5/4/25	190	−50	III
5/25/25	11,941	−2037	
6/1/25	6251	−1568	
7/20/25	1	−1	IV
7/27/25	614	−166	

- **Prompt:** Create a website on the Pi with a button to click to cause the drone to take off and hover.
- **Prompt:** Now add some functionality to the webpage. Add a map with the drone location on it. Use the MAVLink GPS messages to place the drone on the map.
- **Prompt:** Now add the following functionality to the webpage: the user can click on the map, and the webpage will record the GPS coordinates of the map location where the user clicked. Then it will send a "guided mode" fly-to command over MAVLink to the drone.
- **Prompt:** Create a single `.sh` file to do the entire installation, including creating files and directory structures.

The initial prototype was basically functional. Amazingly, after only the first few prompts, the AI offered this:

"Would you like me to explain how any part of this works or help you test it? I can also add features like:

- *Ability to set altitude for each waypoint*
- *Multiple waypoint planning*
- *Return-to-home functionality*
- *Geofencing boundaries*

This unrequested, volunteered feature demonstrated the model had an intimate knowledge of drones and could suggest additional features based on this knowledge. We were surprised and delighted by this suggestion, which proved immediately the concept and power of AI for robot programming.

The LLM was asked to write a shell script to install all of the files needed onto a fresh Linux instance, including the HTML files, Javascript, Linux Systemd service files, Python code, environment setup, etc.

However, after about a dozen prompts, including several where the model stopped, the conversation came to an abrupt end because of the number of tokens in the model was limited. The actual message was *This conversation has reached its maximum length.*

In order to continue development, we created two new conversations (i.e., context windows), but both quickly reached the maximum length without fully functioning code with new requested features. The second conversation even created a prompt for the third conversation to start from. Even given this restriction, the model was able to create detailed user manual, complete with troubleshooting guide and safety guidelines!

At the end of sprint 1, the basic prototype website would load but it was not functional enough to be tested in flight. At that point the lack of memory (number of tokens) was an issue that made further development of the project infeasible with that model's limits at that time, using a browser to copy and paste a single install script, so the project was temporarily abandoned. So impressive as it was, further AI development of this project was halted. AI seemed to have reached its limits.

The unofficial industry standard (more a rule of thumb, not hard and fast) at the time was approximately 1,000 lines of code max for an LLM generated file[14,15]. The install script was 2000 lines of code, so we were clearly at the limit. Furthermore, the install script used multiple languages in one file (Python, HTML, Javascript, CSS, bash), which is not an optimum strategy. It was only chosen for simplicity of evaluation in the first phase of the project. The estimated number of man-hours was 16 h, or two business days. Many (well over half) of those hours were spent waiting for the AI to return a response to the prompt.

Sprint 2: Gemini 2.5 On 25 March 2025, Google announced a public beta of AI programming LLM Gemini 2.5 with 1M tokens. (Claude in sprint 1 was 200k tokens). Excited by the additional capability of the LLM, we started the project again. We continued with the same in browser, single bash install script approach as in Claude: The prompt was asked to write a shell script to install all of the files needed onto a fresh Linux instance, including the HTML files, Javascript, Linux Systemd service files, Python code, environment setup, etc.

In this sprint, we further guided the LLM to develop functionality and tested it on the bench at each iteration. We used five context windows, each

with around 500k tokens (tracked by the UI Gemini Studio). We found after 500k tokens the LLM lost memory of the early parts and necessitated starting a new context window. Each conversation had about 25 individual prompts, some asking for new features, but most reporting a bug in the code during install or on the bench testing which had to be fixed by the LLM.

There were many bugs identified during this process. The debugging procedure we used was to cut and paste the error message into the LLM and ask it to fix whatever bug caused the error message. We did not have the time to go through the code line by line to try to find the bug manually, nor would we want to, since that goes against the spirit of this project.

One of the largest bugs was getting the syntax of an HTML file inside a bash shell script: The shell script would create an HTML file and write the contents of the HTML file from the script. Surprisingly, this syntax was not correctly coded, and this occurred over and over.

About halfway through this sprint, we had a functional script that would install on a fresh OS install on a Raspberry Pi companion computer on the drone, and we were able to demonstrate it in test flight #1. However, it still did not properly update the position on the map.

Toward the end of this sprint, the single script install file was still too unwieldy for the LLM, and we asked it to create several individual files (e.g., one HTML file, one CSS file, one Python file, etc.) These were cut and pasted into an editor and saved individually, an inefficient process.

Although the history was not recorded precisely, we estimate about 30 total hours of this sprint, mostly sending a new prompt, waiting for the result, cutting and pasting it into Linux, running the shell install script, testing the website to control the drone on the bench, and iterating.

At the end of this sprint, the project was still not fully functional, and was not fully demonstrated in a flight test.

Phase II: IDE (Sprint 3) Even though the context window was large enough, the copy/paste approach proved unwiedly, especialy as the project had grown to many individual files, many in different langauges. Therefore in Phase II, we moved an AI integrated into an IDE. This enabled the AI to focus on only certain aspects (e.g., color of a button) at a time. This strategy allowed the context window limitation to be superceded by focusing each prompt on a limited part of the code base.

Sprint 3: Cursor IDE Recognizing that the project needed multiple files (Python, HTML, JavaScript, bash, etc.), we moved to Cursor IDE, a fork of VS Code. This also enabled easy syncing of each version with GitHub. Cursor has its own internal memory locally and can also access various models, including Gemini, ChatGPT, Claude, etc. In addition, having a local IDE allowed testing of basic functionality of the code such as deploying a website on the local machine and local testing of the Python code.

About halfway through this sprint, we had a version of the code that was ready for a real flight test. The first fully successful flight test demonstrated the desire functionality: Mode change, takeoff, land, RTL, plot drone on map in real time, click to fly to a position on the map, and arm/disarm.

The process was not without issues. During the process, we tried Claude, ChatGPT, and Gemini. We occasionally switched when one model repeatedly failed on the same task. One simple code change we requested was the ability to specify the IP address of a drone to control as a UI entry on the webpage, rather than hard coded. The challenges here were surprising, and we believe it was because each individual file was still too large and the model did not have the context of all of the files so was unable to track how changes in one segment of code impacted that of other code segments.

Another common challenge was for the LLM to keep track of the dynamics of the drone update information, from the drone, to the back-end, to the front-end, to make the timing efficient, and to handle transient events such as the drone connecting/disconnecting, being in flight and armed vs. on the ground, etc. This indicated that the project had grown beyond the scope of LLM contextual memory.

In order to address this, we asked the LLM to refactor to compartmentalize all the files to smaller-size files. In practice, this generated a lot of extra bugs that did not work. So, it seems from user perspective that it was saturated.

This sprint used about 30 man-hours of prompt engineering and about 35k lines of code written or rewritten "lines of agent", see the appendix, and 51 GitHub commits.

Phase III: Self test coding (Sprint 4) In Phase III, we developed a testing based approach: The LLM was given a specific function that the code should be able to carry out, and the LLM was tasked with adjusting the code iteratively until the test was met. This removed the need for human intervention and human testing, and dramatically improved the outcome. We describe the detail of this in sprint 4.

Sprint 4: Windsurf IDE In the 4th and final sprint, we switched to Windsurf IDE, a competing fork of VS Code. We hypothesized that it could have more memory capacity awareness of larger codebases vs Cursor. Although both IDEs are proprietary and competitors, Windsurf Cascade was advertised as an agentic experience, ideal for AI-driven multi-file workflows.

We successfully used Windsurf to complete the code refactor, add functionality such as added voice notifications, and add the ability to an pick IP address on the website. This sprint used about 30 man-hours of prompt engineering, about 14k lines of code written or rewritten, and 41 GitHub commits. We also created releases, and had 1.4–1.8 releases on GitHub. A flight test showed enhanced version 1.8 functionality, but the take-off command was not working properly.

A final sprint of 8 h (and 44 more GitHub commits) in the end led to the full version 2.0. This was comprehensively tested on a desktop/cloud and Raspberry Pi companion computer on the drone, on the bench, and fully tested fully in the air. The dates of Phase II-IV and GitHub additions and deletions are summarized in Table 1, and fully documented in Supplementary Information.

Adding it up: Man(person) hours, Machine time The total # of Man(person) hours was approximately 100, or about 2.5 weeks. (The detailed statistics available of all the coding are given in the Supplementary Information) The total # of GitHub commits was 120. The total number of lines of code for the project (version 2.0) is around 10k.

Comparison to CloudStation This project of code is almost identical to a similar project we developed over the last 4 years called called *CloudStation*[9,10] That project had four major releases. The first release was the result of six undergraduates working for two quarters. The second release was the result of three graduate students working for two quarters. The third release was the result of one graduate student working for 2 quarters. The fourth release was the result of the author working for maybe 1–2 weeks. If we estimate the class credit as 3 h of contact time and 6 h of work per student per week, the math comes to about 2000 h. (One quarter = 10 weeks, so 90 h). Thus, this paper demonstrates approximately 20× less hours of work for a very similar result. It would not make sense to compare to other open-source GCS projects such as Mission Planner and QGroundControl, since they have much larger functionality built in, so such a comparison would not be apples to apples. This comparison is for one project only, and this single data point does not constitute a robust statistical benchmark.

We next turn to industry-standard development metrics. We estimate the development time from the Cost of Code Model II, developed at the University of Southern California, and considered the industry standard prior to the AI era[16,17]. The key prediction is the number or person months PM needed to complete a project with size klines of code is given by $PM = A \times (\text{size}) \times b$, where A and b are constants depending on the project details. We take a typical value of 2.94 for A and 1.15 for b, and find an estimate of 93 person months (14k hours). This is about an order of magnitude larger than the hand coded 2000 h used in CloudStation. However, as that was an undergraduate project and not professionally developed, this is understandable. The reduction from 2000 h hand coded to 100 h AI coded (this work), while only one data point, demonstrates the potential improvement in efficiency of AI generated code for drone control. Note that neither of these includes rigorous, validated, certified testing, only a basic level of flight testing under a full suite of standard cases.

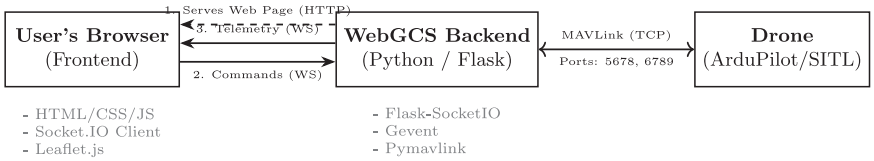

Fig. 4 | Architecture. Three-tier design: the user's browser (frontend), the WebGCS backend (Python/Flask), and the drone. The backend serves the web application over HTTP (1), receives commands from the browser over WebSocket (2), and streams telemetry to the browser over WebSocket (3). The backend communicates with the drone via MAVLink over TCP. Technologies used in each tier are indicated.

Result #2 of 2: "The Software Architecture Itself" (Web based command and control station in the sky)

The second major result of this paper is the software architecture itself: A web based command and control station in the sky. As we mention the introduction, a web based command and control station on the ground was already demonstrated by us[9,10]. This paper "reproduces" those results using generative AI in an extremely efficient, non-human coded result ("The Process").

However, the deployment of a web based control station in the sky had not been demonstrated before, and removes the need for anything other than a web browser to pilot a drone ("The Software Architecture Itself").

Air based control station: website in the sky

On a drone, higher level code is normally run on a so-called "companion computer", since it is the higher level control function of the drone, leaving the lower level functionality (such as self-leveling and stability) to the "low level" on board computer, the flight controller.

In Fig. 2A, we show the existing state of the art of companion computers, both hardware, OS, and software. The software (until this work) is essentially self contained on the drone, with communications to the cloud for additional functionality rare. Importantly, all of the existing software was "hand written", line by line, by *humans*. For example, the most common companion computer codebase with higher level functionality is the so-called "Robot Operating System 2"[4], which has over 2.5 million lines of *human* written code.

In this work, agentic AI large language models (LLMs) create an on board computer drone control station, which runs on a Linux machine on board the drone (in our case, a Raspberry Pi 2 W). The drone control station in the air has the same functionality of the GCS in the first result, but it is in the air. In addition, in contrast to the existing software, it hosts it's own website *in the air*, with no need for any special software on the ground other than a web browser (Fig. 2B). All of the code is created by AI. Thus, in contrast to ROS2, and other drone control code in the air written by *humans*, this drone control code was entirely created by a *machine*: A robot control station (in the air) was created by a machine.

Architecture

Figure 4 shows the system architecture. The design follows a three-tier structure: a frontend running in the user's browser, a backend server (WebGCS) implemented in Python with Flask, and the physical drone, which is monitored and controlled via the MAVLink protocol. The following subsections describe each tier, the technologies chosen, and how they interact. *Frontend (user's browser)* The client is a web application built with HTML, CSS, and JavaScript. It provides the pilot interface: map view, telemetry display, and control inputs. Mapping is implemented with Leaflet.js, which supports interactive maps, overlays, and real-time position updates, and integrates well with web standards and Socket.IO. Real-time communication with the backend uses the Socket.IO client library, giving a single abstraction for a persistent, bidirectional channel: the browser sends commands (e.g., take-off, land, go-to-waypoint) and receives live telemetry (position, attitude, battery, status) without polling. This keeps the UI responsive and avoids the latency and overhead of repeated HTTP requests. *Backend (WebGCS)* The server is implemented in Python using Flask as the web framework. Flask handles HTTP traffic (serving the initial page and static assets) and is extended with Flask-SocketIO to add WebSocket support, so the same process serves both traditional HTTP and long-lived Socket.IO connections. Concurrency is handled with Gevent, which provides greenlet-based concurrency and non-blocking I/O. This allows the backend to manage many simultaneous browser connections and a continuous MAVLink stream without blocking the event loop, which is important for real-time telemetry and command handling. MAVLink integration is done with PyMAVLink, the main Python library for MAVLink: it is well maintained, supports the message set used by common autopilots, and handles serialization, parsing, and connection management. The backend therefore acts as a bridge: it receives MAVLink telemetry from the drone, forwards relevant data to the browser over WebSockets, and translates user commands from the browser into MAVLink messages sent to the drone. *Communication flows* The figure illustrates several distinct flows. (1) The backend serves the web application to the browser over HTTP. (2) Commands from the pilot are sent from the browser to the backend over the WebSocket connection (Socket.IO). (3) Telemetry is pushed from the backend to the browser over the same WebSocket channel, enabling live updates of the map and status displays. Between backend and drone, communication is carried by MAVLink over TCP: the backend maintains a TCP connection to the drone (or to a MAVLink-capable proxy), and uses PyMAVLink to send commands and receive telemetry. TCP was chosen for reliable, ordered delivery, which is important for safety-critical commands and consistent telemetry ordering. *Summary of design choices* The architecture centralizes all MAVLink and business logic in a single Python backend, which simplifies deployment and keeps the frontend as a thin client. Flask and Flask-SocketIO allow a single server to handle both HTTP and WebSockets; Gevent supports the concurrency needed for real-time streams; and PyMAVLink aligns with the existing MAVLink ecosystem. The use of WebSockets for browser-backend communication avoids polling and reduces latency for both commands and telemetry, and Leaflet.js provides a standard way to visualize the drone's position and mission on a map. Together, these choices yield a web-based GCS that can deliver real-time control and monitoring with a clear separation between presentation (browser), application logic (Flask backend), and vehicle protocol (MAVLink over TCP).

The overall architecture—the three-tier split, the choice of Flask, Socket.IO, Gevent, PyMAVLink, and MAVLink over TCP—was decided entirely by the AI. No human guidance was given on how to structure the project.

The figure was produced by the same AI from the prompt "*can you make a diagram of the architecture*". The AI therefore both designed the system and generated this diagram from its internal representation of the modules, their interactions, and the roles of the pilot and the drone within the architecture.

The choice of architecture is excellent. The only disadvantage is that Flask is not meant to scale to production of thousands of drones and thousands of instances of WebGCS. However, it was not asked to do that. It was asked to design something for a single drone. If we had to do it ourselves, we would have chosen the same thing.

There are several other proprietary architectures that achieve the same cloud-based web GCS systems, which we reviewed in detail in refs. 9,10. Since they are proprietary, we have no way to know the internal architecture for comparison.

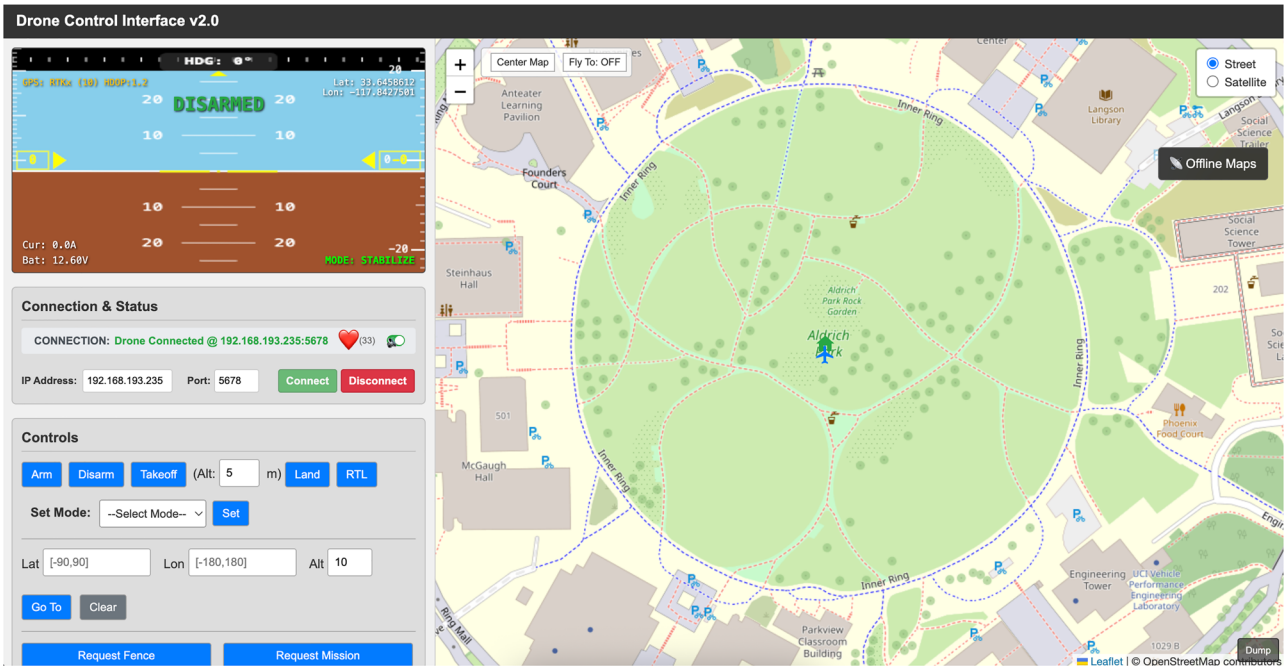

Fig. 5 | Website. The drone location on the map is updated in real time. The HUD on left gives drone information, similar to other GCS software. Control buttons initiate various autonomous actions such as takeoff/land/return to home.

In summary, based on years of experience designing and implementing similar projects, our assessment is that the AI designed architecture is the most appropriate choice for this project (single drone).

Pilot interface: UI. The pilot interface was fully coded by AI. The AI was told to have a map and buttons, and little else. The AI autonomously chose to implement a heads-up display without any specific prompt to do so. We asked the AI to make some cosmetic modifications to the UI and add some debugging features such as logging messages, which it did. The current version graphical user interface (GUI) is shown in Fig. 5.

Performance: flight tests

Flight tests were performed with a small drone with a Raspberry Pi (Fig. 6). Details of the drone are provided in methods and ref. 18. The WebGCS was hosted on the drone, and a laptop was logged onto the drone WiFi hotspot. The connection was stable up to 100 m away, the maximum distance tested.

The initial flight tests show two bugs. One of the times, the drone did not update its position on the map. Another time it did not take off when commanded. The AI was given the bug report and updated to code to fix the bugs. As of version 2.0, there are no known bugs.

The successful flight test consisted of the following sequence of events (Fig. 6), using the website to trigger each event:

- Arm
- Takeoff
- Fly to a point on the map
- Return to launch

Several test flights are shown in Fig. 6, with gradually increasing level of control from WebGCS interface only. During the final test flights, the backup remote control was not used at all. The entire flight was controlled by the AI coded control station of the drone in the air and the website interface.

Although these missions were quite modest and simple, they demonstrated proof of concept and operation. However, this method is easily scalable to much more sophisticated and longer distance missions.

Safety protocols In this work, we flew the drone only over a large field without humans around. We also had a standby remote control with redundant manual control ability if needed. Finally, we implemented a strict geofence in the software.

Simulation and deployment in the cloud In order to enable rapid iteration and testing, a simulation protocol was developed to test the code on a virtual drone in the cloud (Supplementary Fig. S1). This was used continuously during the project development and testing.

Discussion

What is the relationship between the number of tokens in an LLM AI model, and the number of lines of code that model can write, maintain, refactor, enhance, and debug? To our knowledge, there is (as yet) no general answer to this question[19–27]. We discuss the practical experience from this paper, and its implications for the future of AI in robotics below.

In this work, our codebase ended up at about 10k lines of code. This is already quite an accomplishment for an AI code project written completely from scratch with no human lines of code. In our experience, even the best IDEs, LLMs, and models were unable to completely have the entire codebase in memory context sufficient for any given task. For small code changes this was not an issue, for example to change the color of a button. But code changes which required knowledge of the plumbing of the flow of information were more troublesome. In the drone case, the flow and storage of information is one of the primary tasks of the code. Information flows through many modules from the drone, to the computer, to the back end, to the front end, and back again when a pilot requests an action, such as takeoff or land. In our experience, this is about as complex of a codebase as modern AI models can handle, as of the writing of this paper.

Although it depends on the codebase properties, for the purposes of this discussion, we assume 10 tokens is equivalent to one line of code. For large context models, for example, Rando[28] found *The accuracy of Claude 3.5 Sonnet on LongSWE-Bench drops from 29% to 3% when increasing context length from 32K to 256K*. Claude Sonnet is a model marketed for its long-context prowess. As our codebase is around 10k lines of code, we are right at the edge of the scaling of current models. This paper by Rando was not read until after our experience with this project, and so our experience is an unbiased, independent (single case) real world example that is consistent with the more general findings of Rando et al.[28].

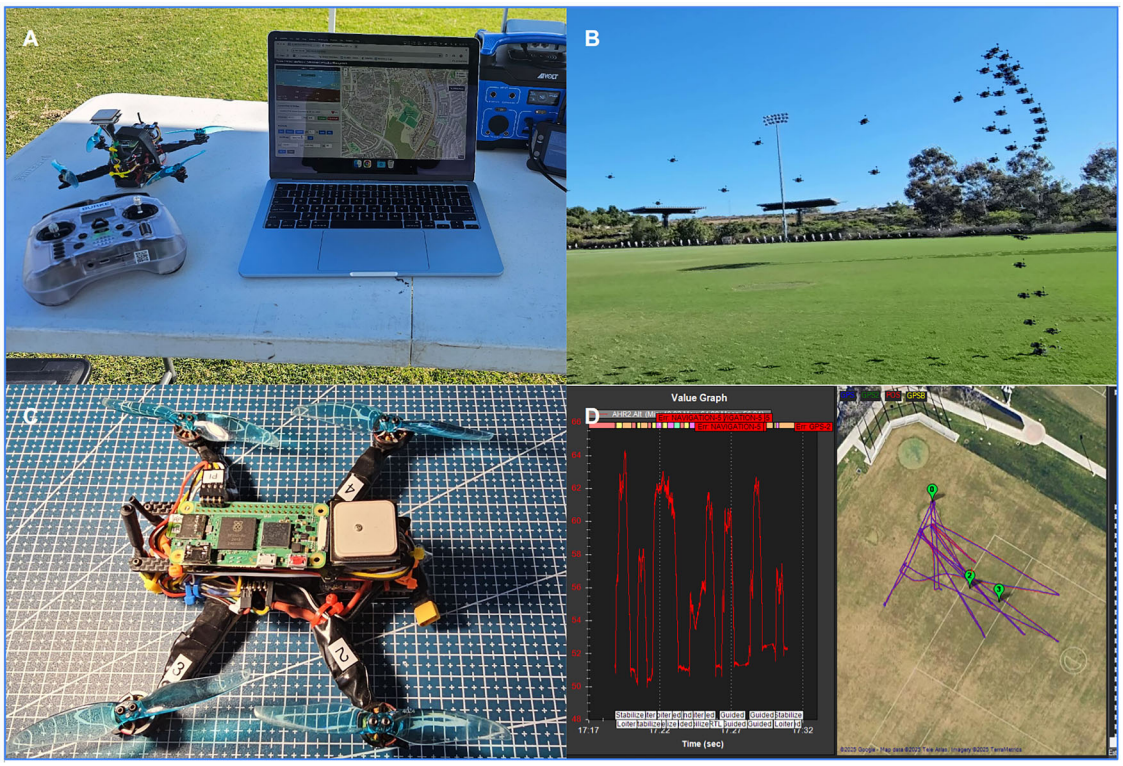

Fig. 6 | Flight demonstrations. A Website seen on the laptop was hosted on the drone. The small handheld radio is a backup and can be used to manually pilot the drone if the WebGCS fails for any reason at any time during the flight. **B** Timelapse image of a flight where only the WebGCS was used for drone control. **C** Picture of the drone. The Raspberry Pi Zero 2 W on the top of the drone hosted the AI generated code. **D** Post flight log analysis showing the drone flight path. The altitude vs. time shows multiple flights, and the map shows the location of the drone during the flights.

As the codebase grew to 10k lines, the context window became a limiting factor, requiring complex refactoring. If the AI cannot load the full context to understand dependencies, how can future maintenance (human or AI) be guaranteed without introducing regressions? There are two possible answers to this. The first is larger context windows. However, more likely the better answer is coordination of agent swarms. Recently (during the review of this paper), Anthropic released a feature called "Claude Code Agent Teams"[29] to do just that. They were able to create a 100k lines of code C compiler that was demonstrated as capable of compiling the entire Linux kernel[30].

A feasible next step would be to extend AI coding up the tech stack to higher order reasoning, such as autonomous navigation based on broad goals. ROS2 in some sense is one of the human written version of that: not just GCS but true autonomy. It is similar to but different from the AI beating the human in a drone race[31], but coding it all with AI *itself*.

Another feasible next step would be using AI coding to go down the tech stack. For example, could one list the requirements of the low level flight control software such as Ardupilot or PX4 as a technical specification and vibe code a new version of Ardupilot? Since Ardupilot codebase has about 1M lines of code, most likely the next step should be some specific task, such as racing, demonstrated by[31].

The next logical steps could be AI agent swarms controlling real drone swarms, or light shows[32]. This is possible but beyond the scope of this work.

Since this project of code is almost identical to a similar project we developed over the last 4 years called called CloudStation, which is openly available on GitHub, it is quite possible that the repo was also used to train the LLMs used. However, the project here had different design choices than CloudStation, including the fundamental webserver (Python Flask instead of Django and Daphne and Nginx, bare metal execution instead of Docker). Thus, based on our experience with CloudStation, the new AI code was not just a clone or copy of it, but genuinely new. Therefore, if the LLMs had been tasked with creating something entirely novel, it still would be possible to achieve this (anecdotal) level of results for code generation for drones.

At the end of this project, we did experiment with Replit, which is advertised to be able to run long production runs, not just single chats. In initial studies, it actually duplicated the exact UI for our project, down to the color choices. Therefore, it seems all the hard work we put into this project is being used by LLMs for other drone projects that replicate this functionality: Our project is now in the training memory of commercially available LLMs, down to the color of the buttons in the UI.

Regarding the *Innovation and Prior Art* for use of LLMs for robots, the prior art can be divided into two categories. The first is using the LLM to write code, but not actually control that robot. The second is using the LLM to actually control the robot. We discuss both next.

There is extensive prior literature on using LLMs in real time for robot control, such as path planning as well as higher level reasoning, reviewed in[33], see also[34–41]. For brevity, we restrict our attention in this section to drones only. The use of LLMs for drone control offers many advantages, as we recently discussed and demonstrated (in a universal, platform independent method that applies to all drones using MAVLink and all LLMs supporting MCP protocol) in[42] and references therein, e.g., [43–79]. Most of the work is recent (from 2025, during which this work was being done), and still under review or preprints. About half of the work is conceptual, but it is clear the advantages of LLMs are well recognized by the larger drone research community. Our work here in this paper is distinct, in that we use LLMs to write the code, but once the code is written, LLMs are not used (i.e., the LLM is not used during flight). For further review of the exciting possibilities of LLMs for drone control *during* flight, we refer the reader to our recent paper[42], and references therein.

In contrast, there is a gap in the literature for LLMs used for code generation of drone GCSs (or drone firmware). To our knowledge, there has been no prior art on LLM coded GCSs, or web based GCSs. Our work in this paper also extends this to enabling a website in the sky, which also is not

present anywhere in prior art or literature. Therefore, this article represents the first demonstration of this application of LLMs in drones: LLM is used in advance to generate deterministic code executed during flight.

Although the generation method is novel, the resulting architecture (Flask, PyMAVLink, WebSocket) is quite standard for this type of application. In this paper, the innovation lies strictly in the process (Generative AI) rather than the product (the software architecture itself), which mimics existing human-designed patterns. In addition, the innovation lies in the hosting of the ground-control station on the drone itself, which has never been demonstrated before, creating a flying website in the sky with no need for any ground based computer power other than a web browser, which can be done on any portable computer or even smartphone or tablet, from anywhere in the world.

Some people may argue that the flight controller is the "brain" of the drone, including the firmware and hardware. In the context of real brains, we argue that the portion of the brain for automatic control, such as breathing, cannot be considered to be the entire "brain". The part of the brain that controls breathing is critical for sustaining life, but is done automatically even during sleep. Most people would not call the brain just the part responsible for regulating breathing. Similarly, the part of the brain that controls gait (i.e., balance during walking and running) is mostly done without effort or "thought". It is part of the brain, but only the lowest level. The "real" brain is usually meant to be higher level thought and reasoning, such as inference, creativity, decision making, and synthesis of multiple sensory inputs together with historical training. The flight controller really only does the analog of "balance" (e.g., keeps the drone level). And even then, there are some flight modes where the flight controller does not even self level the drone. So in the context of drones, it is the higher level command and control software that should be considered the "brain". Typically this is not done on the drone itself, but rather by a human pilot. In this manuscript, the higher level control interface is closer to the "brain" of the drone than the flight controller. In this manuscript also, this critical component is demonstrated on board the drone itself, rather than on a computer on the ground. However, even in this manuscript, a human has to log onto the control panel (a website hosted in the sky) to make decisions, so we have not demonstrated a complete "brain": There is still a human in the loop. The next level up is to have an LLM control the drone through the interface. We recently demonstrated that, in a separate paper[42], and we refer the reader to that manuscript for further discussion of the definition of "brain" in the context of drones.

We conclude it to be currently feasible to use AI to write drone code. We have clearly demonstrated that in flight in this paper. The current limitations are the context window limited the lines of code to around 10k. This limits the functionality that can be implemented. To reach this conclusion, we experimented with several different LLMs with different context windows and, consistent with recently published literature that came out during this work, found that the LLMs begin to fail above a certain number of lines of code that depends on the context window. In conclusion, we have demonstrated a machine making a robot's (drone's) control station, part of the robot's (drone's) "brain".

Methods

Software

The following models were used: Google Gemini 2.5. ChatGPT 4.0. Claude Sonnet 3.5, 3.7. The following IDEs were used: VS Code, Cursor, Windsurf. The code was developed primarily on an AMD Linux laptop, an Intel i9 desktop, and a MacBook Air M4. The drone firmware was Ardupilot, ArduCopter version 2.6.

Hardware

The drone was a sub-250g 4" drone with ELRS radio connections. The drone was powered by a 2S LiPo battery. A Raspberry Pi Zero 2 W was connected to one of the UARTs of the flight controller (Matek F405 Wing, powered by an STM32 F4 based microcontroller) and also powered by one of the 5V BECs on the flight controller. The total current of the avionics at idle throttle was 0.45 A. About 0.1 A of this was for the Raspberry Pi. Details of the build (bill of Materials) are in ref. 80 and ref. 18.

AI

Obviously AI was used to write the code. The only other place AI was used in this work was to fine tune the abstract verbiage from a draft abstract, and to assist in the LaTeX typesetting. Other than that all work including writing this manuscript was done by a human, line by line, word by word, the old fashioned way.

Data availability

All data is included in the manuscript and/or supporting information.

Code availability

The WebGCS code is publicly available at https://github.com/PeterJBurke/webgcs.

Received: 4 September 2025; Accepted: 2 April 2026;
Published online: 15 April 2026

References

1. Team, A. D. Ardupilot: Open source autopilot system supporting multi-copter, fixed-wing, VTOL and more. Version 4.x, accessed 1 August 2025. https://ardupilot.org (2024).
2. Team, P. D. Px4 autopilot. Accessed 1 August 2025. https://px4.io (2025).
3. Team, B. D. Betaflight. Accessed 1 August 2025. https://github.com/betaflight/betaflight (2025).
4. Macenski, S., Foote, T., Gerkey, B., Lalancette, C. & Woodall, W. Robot operating system 2: design, architecture, and uses in the wild. *Sci. Robot.* **7**, eabm6074 (2022).
5. Oborne, M. Mission planner. Accessed 1 August 2025. https://ardupilot.org/planner/ (2025).
6. Team, Q. D. Qgroundcontrol. Accessed 1 August 2025. https://docs.qgroundcontrol.com (2025).
7. ArduPilot Development Team. RFD900 radio modems. Accessed 1 August 2025. https://ardupilot.org/plane/docs/common-rfd900.html (2023).
8. Meier, L. Mavlink: Micro air vehicle communication protocol. Accessed 1 August 2025. https://mavlink.io (2013).
9. Hu, L. et al. "Cloudstation:" a cloud-based ground control station for drones. *IEEE J. Miniaturization Air Space Syst.* **2**, 36–42 (2020).
10. CloudStationTeam. cloud_station_web: Web interface for cloud-based ground control station. Accessed 1 August 2025. https://github.com/CloudStationTeam/cloud_station_web (2024).
11. Guinness World Records. The farthest distance to control a commercially available unmanned aerial vehicle (UAV) at 18,411 kilometers (11,440 miles) (2022). https://engineering.uci.edu/news/2023/2/burke-achieves-distance-world-record-piloting-drone-through-internet. Record held by Peter Burke, confirmed December 9, 2022.
12. Axios. Anthropic's Claude adds new memory features. Axios. Accessed 9 February 2026. https://www.axios.com/2025/10/23/anthropic-claude-memory-subscribers (2025).
13. Anthropic. Claude. Accessed 1 August 2025. https://www.anthropic.com/index/claude (2024).
14. Hansson, E. & Ellréus, O. Code correctness and quality in the era of AI code generation: Examining ChatGPT and GitHub Copilot (2023).
15. Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship* (Pearson Education, 2009).
16. Boehm, B. Safe and simple software cost analysis. *IEEE Softw.* **17**, 14–17 (2000).
17. Boehm, B. W. et al. *Software Cost Estimation with COCOMO II* (Prentice Hall, 2000).

18. Shah, A. S. et al. Detect and avoid (daa) with broadcast remote ID (RID). TechRxiv, preprint under review at *IEEE Aerospace and Electronic Systems Magazine.* 1–12 (IEEE, 2026).
19. Chen, M. et al. Evaluating large language models trained on code. *arXiv preprint* https://doi.org/10.48550/arXiv.2107.03374 (2021).
20. Fan, A. et al. Large language models for software engineering: survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)* 31–53 (IEEE, 2023).
21. Beer, R. et al. Examination of code generated by large language models. *arXiv preprint* https://doi.org/10.48550/arXiv.2408.16601 (2024).
22. Bogomolov, E. et al. Long code arena: a set of benchmarks for long-context code models. *arXiv preprint* https://doi.org/10.48550/arXiv.2406.11612 (2024).
23. Hua, T. et al. Researchcodebench: benchmarking LLMs on implementing novel machine learning research code. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://doi.org/10.48550/arXiv.2506.02314 (2025).
24. Assogba, Y. & Ren, D. Evaluating long range dependency handling in code generation models using multi-step key retrieval. *arXiv preprint* https://doi.org/10.48550/arXiv.2407.21049 (2024).
25. Hsieh, C.-P. et al. Ruler: what's the real context size of your long-context language models? *arXiv preprint* https://doi.org/10.48550/arXiv.2404.06654 (2024).
26. Zhang, X. et al. Infinity bench: Extending long context evaluation beyond 100k tokens. In *Proc. 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* 15262–15277 (Association for Computational Linguistics, 2024).
27. Yen, H. et al. Helmet: How to evaluate long-context models effectively and thoroughly. In *The Thirteenth International Conference on Learning Representations* (ICLR, 2025).
28. Rando, S. et al. Longcodebench: evaluating coding LLMs at 1m context windows. *arXiv preprint* https://doi.org/10.48550/arXiv.2505.07897 (2025).
29. Orchestrate teams of Claude code sessions. Claude Code Documentation, accessed 10 February 10. https://code.claude.com/docs/en/agent-teams (2026).
30. Carlini, N. Building a c compiler with a team of parallel claudes. Anthropic Engineering Blog, accessed February 10, 2026. https://www.anthropic.com/engineering/building-c-compiler (2026).
31. Kaufmann, E. et al. Champion-level drone racing using deep reinforcement learning. *Nature* **620**, 982–987 (2023).
32. Jin, H. et al. From LLMS to LLM-based agents for software engineering: a survey of current challenges and future. *arXiv preprint* https://doi.org/10.48550/arXiv.2408.02479 (2024).
33. Wang, J. et al. Large language models for robotics: opportunities, challenges, and perspectives. *J. Autom. Intell.* **4**, 52–64 (2025).
34. Singh, I. et al. Progprompt: generating situated robot task plans using large language models. *arXiv preprint* https://doi.org/10.48550/arXiv.2209.11302 (2022).
35. Liang, J. et al. Code as policies: language model programs for embodied control. 2023 IEEE International Conference on Robotics and Automation (ICRA) (IEEE, London, United Kingdom, 2022).
36. Ravichandran, Z., Robey, A., Kumar, V., Pappas, G. J. & Hassani, H. Safety guardrails for LLM-enabled robots. *IEEE Robotics and Automation Letters*. Vol 11, 4649–4656 (IEEE, 2026).
37. Jin, Y. et al. RobotGPT: Robot manipulation learning from ChatGPT. *IEEE Robot. Autom. Lett.* **9**, 2543–2550 (2024).
38. Vemprala, S. H., Bonatti, R., Bucker, A. & Kapoor, A. ChatGPT for robotics: design principles and model abilities. *IEEE Access* **12**, 55682–55696 (2024).
39. Cui, C. et al. A survey on multimodal large language models for autonomous driving. In *Proc. IEEE/CVF Winter Conference on Applications of Computer Vision* 958–979 (IEEE, 2024).
40. Chen, Y. et al. Robogpt: an LLM-based long-term decision-making embodied agent for instruction following tasks. *IEEE Transactions on Cognitive and Developmental Systems* (EEE, 2025).
41. Hwang, Y., Sato, A. J., Praveena, P., White, N. T. & Mutlu, B. Understanding generative AI in robot logic parametrization. *arXiv preprint* https://doi.org/10.48550/arXiv.2411.04273 (2024).
42. Ramos-Silva, J. N. & Burke, P. J. A universal large language model—drone command and control interface. *arXiv* https://doi.org/10.48550/arXiv.2601.15486 (2026).
43. Ak Kanigur, N., Mert, M. & Duru, I. Leveraging large language models and artificial intelligence for UAVs in 6G-enabled non-terrestrial networks. In *2025 9th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)* 1–7 (IEEE, 2025).
44. Ping, Y. et al. Multimodal large language models-enabled UAV swarm: towards efficient and intelligent autonomous aerial systems. IEEE Wireless Communications. Vol. 33, 89 –97 (IEEE, 2026).
45. Chagas, F. S., Ruseno, N. & Bechina, A. A. A. Artificial intelligence approaches for UAV deconfliction: a comparative review and framework proposal. *Automation* **6**, 54 (2025).
46. Yang, Z. et al. AI-driven safety and security for UAVs: from machine learning to large language models. *Drones* **9**, 392 (2025).
47. Tian, Y. et al. UAVS meet LLMs: overviews and perspectives towards agentic low-altitude mobility. *Inf. Fusion* **122**, 103158 (2025).
48. Kheddar, H., Habchi, Y., Ghanem, M. C., Hemis, M. & Niyato, D. Recent advances in transformer and large language models for UAV applications. *arXiv preprint* https://doi.org/10.48550/arXiv.2508.11834 (2025).
49. Chen, Y. et al. When large language models meet UAVs: how far are we? *arXiv preprint* https://doi.org/10.48550/arXiv.2509.12795 (2025).
50. Wu, J., You, H., Sun, B. & Du, J. LLM-driven Pareto-optimal multi-mode reinforcement learning for adaptive UAV navigation in urban wind environments. *IEEE Access,* Vol 13, 1520–1535 (IEEE, 2025).
51. Sapkota, R., Roumeliotis, K. I. & Karkee, M. UAVS meet agentic AI: a multidomain survey of autonomous aerial intelligence and agentic UAVs. *arXiv preprint* https://doi.org/10.48550/arXiv.2506.08045 (2025).
52. Cidjeu, D. D., Fendji, J. L. K. E., Kamla, V. C. & Tchappi, I. UAV leveraging GenAI/LLMs, a brief survey. *Procedia Comput. Sci.* **265**, 382–389 (2025).
53. Zhang, X. et al. Logisticsvln: vision-language navigation for low-altitude terminal delivery based on agentic UAVs. *2025 IEEE 28th International Conference on Intelligent Transportation Systems (ITSC)* (IEEE: Gold Coast, Australia, 2025).
54. Yuan, L. et al. Next-generation LLM for UAV: from natural language to autonomous flight. *arXiv preprint* https://doi.org/10.48550/arXiv.2510.21739 (2025).
55. Javaid, S., Fahim, H., He, B. & Saeed, N. Large language models for UAVs: current state and pathways to the future. *IEEE Open J. Veh. Technol.* **5**, 1166–1192 (2024).
56. Yao, F., Yue, Y., Liu, Y., Sun, X. & Fu, K. Aeroverse: UAV-agent benchmark suite for simulating, pre-training, finetuning, and evaluating aerospace embodied world models. *arXiv preprint* https://doi.org/10.48550/arXiv.2408.15511 (2024).
57. Duvvuru, V. S. A., Zhang, B., Vierhauser, M. & Agrawal, A. LLM-agents driven automated simulation testing and analysis of small uncrewed aerial systems. *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. (IEEE, Ottawa, ON, Canada, 2025).
58. Wang, H., Chen, Z., Li, G., Ma, B. & Li, C. Chat with UAV–Human-UAV interaction based on large language models. *arXiv preprint* https://doi.org/10.48550/arXiv.2512.08145 (2025).
59. Khan, A.-M. et al. Context-aware autonomous drone navigation using large language models (LLMs). In *Proc. AAAI Symposium Series* Vol. 6, 102–107 (AAAI Press, 2025).
60. Han, B., Chen, Y., Li, J., Li, J. & Su, J. Swarmchain: Collaborative LLM inference for UAV swarm control. *IEEE Internet of Things Magazine*. Vol. 8, 44–51. (IEEE, 2025).

61. Eumi, E. M., Abbass, H. & Marcus, N. Swarmchat: An LLM-based, context-aware multimodal interaction system for robotic swarms. In *International Conference on Swarm Intelligence* 181–192 (Springer, 2025).
62. Schuck, M. et al. SwarmGPT: combining large language models with safe motion planning for drone swarm choreography. *IEEE Robotics and Automation Letters. Vol.* 10, 215–222. (IEEE, 2025).
63. Nunes, D., Amorim, R., Ribeiro, P., Coelho, A. & Campos, R. A framework leveraging large language models for autonomous UAV control in flying networks. *2025 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)* (IEEE, Nice, France, 2025).
64. Mishra, S. et al. Aermani-VLM: Structured prompting and reasoning for aerial manipulation with vision language models. *arXiv preprint* https://doi.org/10.48550/arXiv.2511.01472 (2025).
65. Koubaa, A. & Gabr, K. Agentic UAVS: LLM-driven autonomy with integrated tool-calling and cognitive reasoning. *arXiv preprint* https://doi.org/10.48550/arXiv.2509.13352 (2025).
66. Lim, S. K., Chong, M. J. Y., Khor, J. H. & Ling, T. Y. Taking flight with dialogue: Enabling natural language control for PX4-based drone agent. *arXiv preprint* https://doi.org/10.48550/arXiv.2506.07509 (2025).
67. Ahmmad, S., Aditto, Z. A., Hossain, M. M., Yeasmin, N. & Hossain, S. Autonomous navigation of cloud-controlled quadcopters in confined spaces using multi-modal perception and LLM-driven high semantic reasoning. *arXiv preprint* https://doi.org/10.48550/arXiv.2508.07885 (2025).
68. Choutri, K. et al. Leveraging large language models for real-time UAV control. *Electronics* **14**, 4312 (2025).
69. Chen, G., Yu, X., Ling, N. & Zhong, L. Typefly: flying drones with large language model. *arXiv preprint* https://doi.org/10.48550/arXiv.2312.14950 (2023).
70. Zhao, J. & Lin, X. General-purpose aerial intelligent agents empowered by large language models. *arXiv preprint* https://doi.org/10.48550/arXiv.2503.08302 (2025).
71. Cleland-Huang, J. et al. Cognitive guardrails for open-world decision making in autonomous drone swarms. *arXiv preprin* https://doi.org/10.48550/arXiv.2505.23576 (2025).
72. Navarro, A., de Quinto, C. & Hernández, J. A. Beyond visual line of sight: UAVs with edge AI, connected LLMS, and VR for autonomous aerial intelligence. *arXiv preprint* https://doi.org/10.48550/arXiv.2507.15049 (2025).
73. Moraga, Á, de Curtò, J., de Zarzà, I. & Calafate, C. T. AI-driven UAV and IoT traffic optimization: Large language models for congestion and emission reduction in smart cities. *Drones* **9**, 248 (2025).
74. Wassim, L., Mohamed, K. & Hamdi, A. Llm-daas: Llm-driven drone-as-a-service operations from text user requests. In *The International Conference of Advanced Computing and Informatics*, 108–121 (Springer, 2024).
75. Majumdar, S., Kirkley, S. E. & Mallik, B. B. LLM-guided hybrid architecture for autonomous fire response: dialog-driven planning in space and disaster missions. In *Proc. IEEE World AI IoT Congress (AIIoT)* 1049–1054 (IEEE, 2025).
76. Zhou, Q. et al. LLM-QL: a LLM-enhanced Q-learning approach for scheduling multiple parallel drones. *IEEE Transactions on Knowledge and Data Engineering. Vol* 37, 1102–1115. (IEEE, 2025).
77. Chen, G., Yu, X., Ling, N. & Zhong, L. Chatfly: Low-latency drone planning with large language models. *IEEE Transactions on Mobile Computing. Vol.* 24, 1890–1904. (IEEE, 2025).
78. Tazir, M. L., Mancas, M. & Dutoit, T. From words to flight: Integrating OpenAI ChatGPT with PX4/Gazebo for natural language-based drone control. In Proc. *2023 International Workshop on Computer Science and Engineering*. 45–52 (Springer, 2023).
79. Xiao, J., Tsao, C. W., Zhang, Y. & Feroskhan, M. FM-planner: foundation model guided path planning for autonomous drone navigation. *arXiv preprint* https://doi.org/10.48550/arXiv.2505.20783 (2025).
80. UCI Drone. RotorBuilds, accessed 1 August 2025. https://rotorbuilds.com/build/33054 (2025).

Acknowledgements

The author thanks Shanyu Mou and Bogdan Kovtun for flight spotting. This study received no funding.

Author contributions

PJB conceptualized and supervised the project, designed and built the drone, ran the AI prompts to write the code, tested the code, did the flight test, did the analysis, and wrote the paper.

Competing interests

The author declares no competing interests.

Additional information

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s44387-026-00101-6.

Correspondence and requests for materials should be addressed to Peter J. Burke.

SUPPLEMENTARY MATERIALS

Performance: Simulation and deployment in the cloud

Ardupilot has a mature drone simulator program that includes physics[81]. We have developed an install script to set it up in a Linux cloud instance and run as an "always on" service : A virtual drone in the cloud[82]. Without firewalls, any pilot can fly the virtual drone, using the IP address of the cloud Linux instance to connect. The pilot would need Mission Planner or QGroundControl installed on their local PC. With a firewall, a good option is to create a VPN such as Zero Tier. In addition, WebGCS can be installed on *another* Linux instance in the cloud. Again, with this setup, the drone pilot can log onto the WebGCS server, this time using only a web browser, without the need for a local installation of any other software. In the WebGCS interface, the IP address of the virtual drone can be entered, allowing testing of using WebGCS to fly the virtual drone, all without the need for any software on the local machine. This is an excellent way to test the program in a safe way. Detailed video and text tutorials that walk the user step by step through the process are available on YouTube[83, 84] and GitHub[82]. Figure shows a typical simulated drone test architecture.

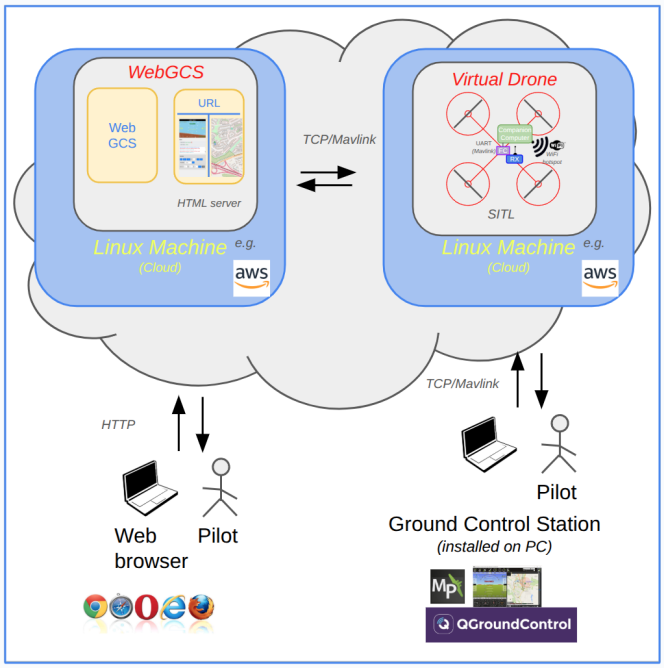

Figure S1. Virtual drone. One (cloud based) Linux machine hosts a virtual drone, which can be controlled with MAVLink/TCP from anywhere in the world with an internet connection. The bottom right pilot logs on using a PC with a ground control station installed on it, such as Mission Planner, or QGroundControl. Another Linux machine (also cloud based) hosts WebGCS, which connects to the virtual drone also through TCP/MAVLink. The pilot on the bottom left only uses a web browser to log into WebGCS and control the virtual drone.

Request for industry

When trying to track the detailed commits, code evolution, and prompt history, we were surprised to find that industry does not have a standard, straightforward method to save a log of the history under a given account. In fact, some companies delete the history after a certain period of time, and there seems to be no public facing industry standard about data retention for consumers. Of the total history, several tracking metrics would be useful to have integrated into the models: Track hours, track number of prompts, exact prompts, number of tokens. A challenge is that the models and even number of tokens are considered proprietary, closely guarded trade secrets, so research in this area is difficult to progress (e.g. number of tokens vs. number of lines of code).

Safety

The general problem of testing AI generated code is an open problem and was not solved comprehensively or rigorously in this work. While this work demonstrates the importance of such an advance in a visceral way, it is beyond the scope of this paper.

Nevertheless, we were able to test the features in a relatively safe environment. Testing it in a simulation environment is a good approach, but when dealing with the physical world, it may deviate from simulations. At this point it is not clear what the correct testing procedure should be if, for example, this is deployed on drones in higher risk operations, for example operations

over people, or beyond visual line of sight. While human code can be debugged and tested line by line, more research needs to be done on the safety and reliability of AI generated code.

"Formal verification": The use of AI for aviation software, and its testing for safety, is an unsolved problem in the literature, and in the industry[85–90]. This is part of a large issue of technical debt in AI generated code[91–97]. Many would argue that reliance on empirical flight testing in a field is insufficient for aviation software, but rather formal verification is required prior to mass deployment[98]. In this paper, we have not resolved the issues of how to effectively test, certify, verify, validate, and standardize aviation software written by LLMs. This is a demonstration, but not a fully tested system ready for deployment at a massive scale.

That said, to our knowledge the industry currently lacks the regulatory and standardization frameworks to manage the risks of stochastic code generation. Current standards were written for a deterministic world. Neither IEEE 1012[99] nor ISO/IEC 42001[100] are capable of handling AI generated code. Verification tools, when state space explosion is considered, are still in their infancy[98].

Stochastic nature of LLMs: Due to the stochastic nature of LLMs, if another researcher inputs our exact prompts, they will likely generate different code. This lack of determinism must be account for by clear tests on the functionality of the code. That is why the component level testing is important. We describe the tests in more detail below.

Auditing by humans: The code was extensively tested, including components and subroutines, during the development. In this sense, it was "audited" by a human. However, it was not line by line audited, where a human read every single line of code to manually inspect for bugs. A common issue in LLM generated code is that manual auditing in many cases negates the efficiency gain of using LLMs in the first place[91–97].

Component testing during development: Here we elaborate on the testing baked into the development procedure in Phase III, mentioned briefly in the main text. Instead of manual, line by line code editing and auditing, each major component was tested one at a time, including testing suggested by the AI. For example, test the establishment and maintenance of connection to the drone. Or test mode change. Or test plot position on a map. These could easily be tested with the simulated drone

We divided the project into logical components and sub-components, and tested the functionality of each individually (human and AI collaborating on the testing together), before combining the entire project into one fully functional system. In fact, this is the same testing regimen we used during the CloudStation project, which was entirely human coded: Different sub sections of the code were developed and debugged by manual tests. In fact our team of six undergraduate students each focused on a different task, similar to how it is done in industry: One engineer is for front end dev, one back end, etc. The entire team (similar to an entire company) must come together in a “team meeting” to ensure all the components are working together. The main point of this work is the reduction in workload so one person could do in 2 weeks what it took a team of 6 to do in 6 months. In that project, we also did not do line-by-line manual review. It was not “certified” or formally verified, it was a proof of concept. Similarly, this work is not “certified” or “formally verified”, but rather a proof of concept and first step towards LLM use in drones, broadly writ. We do also not know how to specify all the edge cases

During the project we had at least 13 different Python test scripts written for testing a specific component or functionality.

Human led safety features specific to drone control stations: In this work, we implemented several safety features by design and architecture specific and unique to drone control stations. This was based on our extensive experience building and flying a large variety of drones and UAVs.

One thing that was manually implemented was a heartbeat sound and icon. This is part of the MAVLink spec, and if the heart beat sound goes away, the pilot immediately knows. Thus, not only is code function important, the response of the human and drone to e.g. lost link must be fully mapped out. In principle, the drone could be programmed to enter an autonomous mode such as land or return to home in case of lost link. We did it one way here, but it is beyond the scope of this paper to cover all use cases and all edge cases.

Hallucinations: We experienced extensive hallucination during the development process. These were due to initially (Phase I) context window size limitations, where the LLM lost memory and hallucinated the state of the code, or the solution. During the later phases (Phase II,III) the hallucinations were obvious and resulted in broken code despite prompts explaining the error. For example, the IP address was hard coded, and when asked to make it a parameter, the LLM still hard coded the IP address. In fact in practice probably about half of the human feedback during the development phase was finding those errors and “coaxing” the LLM to fix them. In Phase III (test based phase), the appropriate approach we developed was to have the LLM self test. That reduced hallucinations significantly, because if the test failed, the LLM would mark it. So there was manual auditing of the code (testing) during the development phase, and the simulated drone was useful for this purpose. During flight, in this paper, there is no LLM interface, so hallucinations are not an issue: The code we demonstrate in this paper is deterministic, rendering the risk of hallucinations during flight a non-issue.

AI history

WebGCS 2025

Sprint 1: Claude

2/23/2025-3/28/2025

200k tokens max

Conversation #1

Controlling a Drone with Python and
MAVLink
Created: 2/23/2025 13:34
Updated: 3/26/2025 22:16
Exported: 7/27/2025 19:23
Link: https://claude.ai/chat/b37c43eb-eab3-466c-833d-bf5f26b9a456

Conversation #2:

Drone Control System for Raspberry Pi
Created: 3/27/2025 12:53
Updated: 7/20/2025 17:37
Exported: 7/28/2025 13:49
Link: https://claude.ai/chat/2eb572ee-2ce3-4c21-bbe7-0d7c7869346b

Conversation #3:

Created: 3/28/2025 0:49
Updated: 7/20/2025 17:37
Exported: 7/28/2025 14:14
Link: https://claude.ai/chat/11de3b81-d087-4c92-9045-40b3c5ad6f18

Sprint 2: Gemini

3/29/2025-4/26/2025
Flight test #1 4/5/2025 (Version 5.11 or 5.12)
No map but buttons worked

Release dates

1 million tokens

Gemini 2.5 was released in stages.

- An experimental version, Gemini 2.5 Pro Experimental, was released on March 25, 2025.
- Preview models of Gemini 2.5 Flash became available on April 17, 2025, with another preview on May 20, 2025.
- The stable versions of Gemini 2.5 Pro and Gemini 2.5 Flash were officially released as Generally Available (GA) on June 17, 2025.
- "Raspberry Pi Drone Control System Update" 3/29/2025
    - 2.31 to 2.45 Raspberry pi install script (3/29/2025)
    - Must have been more from Claude to here, got up to version 2.31 for raspberry pi.
    - 430k tokens
    - Created chat for next conversation
- "Rasbperry Pi Drone Control Script 2" 3/30/2025
    - 2.46 to 2.63
    - 417k tokens
    - Prompt for new chat
- "Drone Control System: Raspberry Pi V5" 3/30/2025
    - "Did development on desktop 3/29/2025 and it works well. Now move it back to host the website on the pi itself."
    - 2.53-5.11 (I skipped to 5.2 for some reason , no 4.0 no 3.0; mavlink router also)
    - 420k tokens
    - Works. (Flight test #1)
- Stats this sprint:
    - 5M tokens
    - 60 prompts/requests
- **********TRAVELBREAK**********
- "Raspberry Pi Drone Control System v5.12" 4/26/2025
    - 97k tokens
    - 5.11-5.12
    - Trying waypoints
    - No workie.
    - Install script 2007 lines

    - Gave up on browser
- “Desktop Drone Control System V 2.64” 4/26/2025
    - Tried to run on desktop instead since pi to unwieldy
    - 471k tokens
    - Started doing files one by one
- Stats this sprint:
    - 1M tokens
    - 20 prompts/requests

It seems I started with Gemini before Mar 29, 2025.
From file:
"model": "models/gemini-2.5-pro-exp-03-25",
I cut and paste the code from the chat window to the raspberry pi.
And then kept doing that. I quickly ran out of tokens and had to have Gemini summarize the results and give instructions for the next version (including the sh code) and then start in a new window.

I asked it to do all in one install script, including a script to install html javascript and css.
Eventually it got overwhelmed.
I lost the chats in Gemini.
But I was amazed that it worked right away, and it suggested things like geofence and waypoints as options (it knew about the concepts already).

Then I switched to Gemini AI Studio on Apr 26, 2025
I guess the reason was to see how many tokens I had left.
At that point it was already version 2.31.
I kept doing that for a while.
Then I had to stop for class when I got a basic one working.
Also it was harder and harder to add features with the cut and paste.
And to have one large install file did not work, it made more sense to have a separate index.html and css and javascript file, and a separate install file, and move to github versioning.
On Apr 26, 2025 I guess that was the last time I used Gemini AI Studio, and had both a desktop and an pi version. Desktop version 2.64. Pi version 5.12. Gemini kept track of the versioning for me in AI Studio website.
On the third file there the comment was
“Did development on desktop 3/29/2025 and it works well. Now move it back to host the website on the pi itself”
https://aistudio.google.com/u/4/library is the way to get google ai studio history.

It looks like I also tried Claude Sonnet on or around 4/26/2025.

Sprint 3: Cursor

May 2, 2025 -June 4, 2025

Sprint 3A:

April 26, 2025 -May 2, 2025
7k lines of code
Github 41 commits

Flight test #2 5/10/2025 Version commit e6698767ab162ab1117ec4dd60b9fd5f1eccf74a
From Cursor.
Everything worked!

Sprint 3B:

May 30, 2025-June 4, 2025
5k lines of code
Github 10 commits

Hours:

Cursor May 2, 2025
9:50 am-6:30 pm 8 hours
May 29, 2025
1:20 pm - 11:17 pm 9 hours
May 30, 2025
4:11 pm-8:05 pm 4 hours
Jun 1, 2025
1:48 pm - 6:42 pm 5 hours
Jun 2, 2025
2:50 pm-4:30 pm please half hour at midnight 2 hours
Jun 4, 2025
8:00 am - 11:00 am 3 hours

Then 7/27/2025 8 hours, cursor, github, to v 2.0 (enhancements & bug fixes)

So at some point after my academic duties were covered I moved to an IDE such as Cursor VS Code etc. Tried also gemini chatgpt and claude for all. Did not work great but got it to work, and integrate with github.
Cursor was April 26, 2025 sprint, and then May 31, 2025 (It was on Mac).

The first commit was Apr 26, 2025 to the github WebGCS repo.
Version v2.63-Desktop-TCP
It must have come from an IDE.
It was based on Desktop version 2.63.
It had separate html css js python files, an install.sh script, and some extra test scripts.

During the process tried on cursor, windsurf ide. Tried claude, chatgpt, gemini. Switched occasionally. Sometimes it got hung up on simple things like trying to change the ip address of the drone from hard coded to a parameter to enter from the website.

Also when I asked it to refactor to compartmentalize all the files to smaller size files, it created a lot of extra bugs that did not work. So it seems from user perspective that it was saturated.

Sprint 4: Windsurf

May 31, 2025 - June 2, 2025
V 1.4-1.8 released.

Statistics:

1095 messages
14k lines of code
41 Github commits
Windsurf/cascade cannot export chat histories:
https://github.com/Exafunction/codeium/issues/127

Details:

May 31, 2025 switched to windsurf/cascade. (It was on Asus R14 laptop).

Version 1.8 was released on Jun 2, 2025.

There is a file on my google drive under webgcs in peter.burke.john@gmail.com called webgcsdevelopment log.

June 1 2025
Finished major refactor.
Added voice notifications.
Added ability to pick IP address on website.
https://github.com/PeterJBurke/WebGCS
Version 1.7 ready to fly!

Flight test #3 7/26/2025
Version 1.8
Some worked, takeoff no.

Sprint 4B: Windsurf July

7/27/2025
Fixed bugs.
Release 2.0
Flight test #4 Success with v 2.0.

Statistics:

Github 44 commits

Hours:

Cursor May 2, 2025
9:50 am-6:30 pm 8 hours
May 29, 2025
1:20 pm - 11:17 pm 9 hours
May 30, 2025
4:11 pm-8:05 pm 4 hours
Jun 1, 2025
1:48 pm - 6:42 pm 5 hours
Jun 2, 2025
2:50 pm-4:30 pm please half hour at midnight 2 hours
Jun 4, 2025
8:00 am - 11:00 am 3 hours
Total cursor hours 31 hours
Probably gemini same
Probably windsurf half
Total 31+31+15=77
77/8=9.625 work days or two work weeks.
Claude pre gemini maybe 1-2 days.

Then 7/27/2025 8 hours, cursor, github, to v 2.0 (enhancements & bug fixes)

Github

92 commits up to 7/26/2025
129 total on 7/28/2025.

Sprint 1:
Mar 25, 2025 when Gemini 2.5 experimental was released (not yet github)

Github documented sprints:
Sprint 2:
Apr 26, 2025
May 2, 2025
May 6, 2025
May 8, 2025

Sprint 3:
May 27, 2025 (Release 1.0)
May 28, 2025
May 29, 2025 (Cursor)
May 30, 2025 (Cursor)
May 31, 2025 (Cursor, Windsurf)
Jun 1, 2025 (Windsurf) (Divided into 5 tasks)
Jun 2, 2025 Release V 1.8
Jun 3, 2025
Jun 4, 2025