ACTION

Atlantic Coastal Tide Instrumentation and Observation Network

Technical Reference Manual

Software, Firmware, and Operations Reference

Revision date: July 18, 2026
Source repository: github.com/cfi1513840/tide-gauge

Table of Contents

1. Introduction

1.1 Purpose of This Manual

This manual documents the software, firmware, and operational architecture of ACTION (Atlantic Coastal Tide Instrumentation and Observation Network) tide and weather monitoring stations. It is intended as a working reference for anyone responsible for keeping a deployed station running -- primarily an on-site IT technician who did not build the system and may not have routine access to the development team, but also useful to anyone extending or modifying the software.

The manual is organized so that the material most likely to be needed in an actual incident -- checking whether the system is healthy, and what to do if it isn't -- comes first, in Section 2. Everything after that builds up the architectural and module-level detail needed for deeper troubleshooting or future development.

1.2 Audience

This manual assumes the reader is technically capable (comfortable with a command line, SSH, and basic Linux system administration) but has no prior familiarity with this specific codebase. It does not assume Python development experience, though some is helpful for anything beyond the operational tasks in Section 2.

The most common real-world scenario this manual is written for: a station (for example, one operated autonomously by a school or institution without a co-located developer) stops updating or otherwise misbehaves, and someone with general IT skills but no history with the project needs to determine what's wrong and, ideally, fix it without needing to reach the original developer.

1.3 System Summary

An ACTION station consists of one or more ultrasonic sensors reporting water-level distance readings via either a LoRa radio link or a Blues Notecard cellular connection, a Raspberry Pi that receives, processes, and stores those readings, and a web-based dashboard (tide plots, weather, and an optional email/SMS alert subscription system) served from that same Raspberry Pi. A single long-running Python process, tide.py, is the heart of the system -- it ingests sensor data, writes to local and time-series databases, generates the web content, and sends alerts, all from one continuously running process managed by systemd.

> **NOTE:** Two genuinely independent data paths feed the system: a LoRa radio link (sensor to a local receiver) and a Blues Notecard cellular link (sensor directly to Blues Notehub.io, then via webhook into tide.py). A station may use either or both, and they do not share any hardware or software path between the sensor and tide.py. See Section 3.2 for the corrected data-flow diagram.

2. IT Technician Operations Guide

This section is deliberately placed before the architectural detail. If the station appears to be down or misbehaving, start here.

2.1 Is the System Actually Running?

The main process runs as a systemd service named tide, started automatically at boot and restarted automatically if it crashes (systemd's default restart behavior applies here, so a full power-cycle recovers on its own without intervention). Check its status first:

```
sudo systemctl status tide
```

A healthy result shows "active (running)" and a recent-looking uptime. If it shows "failed" or "inactive", the service has stopped and needs a restart (Section 2.3).

Two related services may also be present, depending on the station's configuration:

- tidemonitor -- an optional local status-display process, not required for the station's core function
- Any cron-based jobs -- tideplot.py, which generates the tide graph image, typically runs on a schedule via cron rather than as a long-running service; check with: crontab -l -u tide

2.2 Is the Website Actually Updating?

Even if the service shows "active", confirm the public-facing output is current. Open the station's tide page in a browser (the URL is station-specific -- check tide.env's TIDE_URL if unknown) and look at the timestamp on the most recent plotted data point. If it is stale by more than a few minutes beyond the sensor's normal reporting interval, something upstream of the display -- sensor communication, database writes, or plot generation -- is not working even though the main process hasn't crashed outright.

A quick command-line check of the same thing, from the Pi itself:

```
sqlite3 /var/www/html/tides.db "select max(dtime) from sensors;"
```

Compare that timestamp to the current time. A gap larger than the expected reporting interval (typically on the order of a minute) indicates the sensor data pipeline has stalled even if tide.py itself is still running.

2.3 Common Remedial Actions

Restarting the main service

This is the correct first response to almost any "it's misbehaving but I don't know why" situation -- it clears any hung state without needing to diagnose the underlying cause:

```
sudo systemctl restart tide
sudo systemctl status tide   # confirm it came back up cleanly
```

Rebooting the Raspberry Pi

If a service restart doesn't resolve the issue, or if the Pi itself seems unresponsive (SSH hanging, web server not responding at all), a full reboot is the next reasonable step. Because tide is enabled via systemd (WantedBy=multi-user.target), it starts automatically on boot with no manual intervention required:

```
sudo reboot
```

Allow a few minutes after reboot before expecting fresh data -- tide.py's own startup includes a deliberate delay (ExecStartPre=/bin/sleep 180, i.e. three minutes) before it begins running, to give networking and other system services time to come up first.

Checking recent log output

If a restart doesn't resolve the problem, or you want to understand what went wrong before restarting, check the log:

```
tail -100 /home/tide/bin/tidegauge/newtide.log
```

Look for repeated exceptions/tracebacks, especially ones occurring right before the service stopped responding. systemd's own journal is also worth checking, particularly if the service crashed rather than hung:

```
sudo journalctl -u tide -n 200 --no-pager
```

2.4 Remote Access

Two remote access methods are available, depending on what kind of access is needed:

- SSH -- for anything in this section (checking status, restarting services, reading logs). This is the primary tool for routine remote troubleshooting.
- TigerVNC -- a full remote desktop session, useful if a graphical view is needed (for example, the optional tidemonitor.py/tidedisplay.py local display, or general desktop-level troubleshooting). TigerVNC replaced an earlier RealVNC-based setup specifically to avoid an ongoing paid subscription; a TigerVNC server must be running on the Pi and a TigerVNC viewer on the technician's own machine to connect.

> **NOTE:** Neither of these grants access to anything beyond the Pi itself -- if the issue is with an external account or service (Notehub, Brevo, Twilio, Cloudflare -- see Section 10), remote desktop/SSH access to the Pi will not help diagnose or fix it. Escalate per Section 2.6 in that case.

2.5 What NOT To Do Without Escalating

> **CAUTION:** Do not edit tide.env, tide_constants.json, or any of the Fernet/argon2 key files (k1, k2, k3, ku) without understanding their role -- see Section 7. These control encryption and site-specific configuration; an incorrect edit can silently break alert delivery or, in the worst case, make stored user data unrecoverable.
>
> **CAUTION:** Do not manually edit the SQLite database (tides.db) directly unless you are certain of the schema and are working against a backup copy, not the live file -- tide.py accesses this file continuously, and a manual edit at the wrong moment can corrupt it or collide with an in-progress write.

Routine restarts, reboots, and log inspection (Sections 2.3–2.4) are safe to perform without prior authorization. Anything involving configuration files, encryption keys, database schema changes, or code changes should be escalated first.

2.6 Escalation

For issues that aren't resolved by a service restart or reboot, or that involve any of the caution items above, escalate to the system's maintainer rather than attempting further changes. [Ken: insert the actual contact path you want documented here -- e.g., your own contact info, your colleague's, or a shared point of contact, along with any expected response-time expectation for an autonomous site like Lincoln Academy.]

3. System Overview

3.1 High-Level Description

Each ACTION station centers on a Raspberry Pi running tide.py, a long-running Python process that: (1) receives water-level sensor readings, (2) stores them in a local SQLite database and, in parallel, a time-series InfluxDB instance, (3) periodically generates tide/weather plots and web pages, (4) checks user-configured alert thresholds and sends email/SMS notifications when they're crossed, and (5) serves all of the above via a local Apache web server with a small set of CGI scripts handling the alert-subscription workflow (sign-up, login, password management).

3.2 Sensor Data Flow -- Two Independent Paths

A station's sensor data can arrive via either of two completely independent paths, and a given deployment may use one or both simultaneously for different sensors:

Path A: LoRa radio

A CubeCell-based sensor transmitter (running the CubeCell TX firmware sketch, Section 4) reads the ultrasonic distance sensor and transmits over LoRa RF to a separate LoRa32 receiver device (running the LoRa32 RX sketch), which is connected via USB serial to the Raspberry Pi. tideget.py reads the incoming serial data and hands it off into tide.py's main processing loop.

Path B: Blues Notecard cellular

A Blues Notecard (paired with a Swan microcontroller, running the Notecard TX firmware sketch) reads the same kind of sensor and transmits directly over its own cellular connection to Blues Notehub.io -- a cloud service operated by Blues Wireless, entirely independent of the Raspberry Pi or any local radio link. Notehub.io then routes the data via an HTTP webhook to tidenote.py's NotehubReceiver class, which is integrated into tide.py's own main loop via non-blocking select.select() polling (decoupling Notehub's own HTTP delivery cadence from tide.py's sensor-sampling loop).

> **NOTE:** These two paths do not share any component between the sensor and tide.py -- not the receiving hardware, not the transport protocol, not the software module. A station's SENSOR_SOURCE setting in tide.env ('lora' or 'notehub') reflects which path a given deployment actually uses. An earlier version of this manual's architecture diagram incorrectly showed both paths converging through the LoRa receiver; that has been corrected here.

3.3 From Ingestion to Display

Regardless of which path delivered it, incoming sensor data is processed uniformly from that point forward: tidedatabase.py's DbManage class writes it to both the SQLite sensors table and the corresponding InfluxDB measurement; tidealerts.py's TideAlerts class evaluates it against any active user alert thresholds; and tideplot.py (run periodically, typically via cron) reads back the accumulated history to render the tide/weather plot displayed on the station's web page.

4. Hardware and Firmware

4.1 Physical Components

Component	Role
Ultrasonic distance sensor	Measures distance from a fixed mounting point down to the water surface; smaller distance = higher water level
Heltec CubeCell (LoRa path)	Reads the sensor and transmits readings over LoRa RF
Heltec WiFi LoRa 32 (V3) (LoRa path)	Receives LoRa transmissions, outputs to the Raspberry Pi via USB serial
Blues Notecard + Swan (cellular path)	Reads the sensor and transmits readings via cellular to Notehub.io directly
Raspberry Pi (4 or 5)	Runs tide.py and all station software; hosts the local web server

4.2 Firmware Sketches

As of the most recent firmware housekeeping pass, the repository's Arduino sketches (.ino files) were consolidated from six node-specific files down to three general-purpose templates, one per hardware/path combination:

Sketch	Target Hardware	Data Path
SampleCubeCellTX.ino	Heltec CubeCell	LoRa (transmit side)
SampleLoRa32RX.ino	Heltec WiFi LoRa 32 (V3)	LoRa (receive side)
SampleNotecardTX.ino	Blues Notecard + Swan	Cellular (Notecard to Notehub.io)

> **NOTE:** A Blues Notecard/Notehub.io account is a prerequisite for any station using the cellular path -- see Section 10 for external account requirements. The Notecard itself does not appear directly in tide-gauge's Python code; it communicates with Notehub.io independently, and only Notehub's outbound webhook (handled by tidenote.py) is visible to tide.py.

5. Software Architecture

All server-side code is Python, organized as a set of modules imported by tide.py, plus a set of standalone CGI scripts serving the web-based alert subscription workflow. The table below summarizes each module's role; see the module's own source for full detail.

5.1 Core Modules

Module	Role
tide.py	Main orchestrator -- the long-running process managed by systemd. Runs the primary sensor-sampling loop, dispatches to all other modules on a timed schedule (main_loop_count), and integrates the Notehub webhook receiver.
tidehelper.py	Shared Constants class (site configuration, InfluxDB client, sun-times, email/SMS Notify class) loaded once at startup.
tideget.py	Reads sensor data from USB serial (LoRa path) and other external sources (NDBC buoy data, RSS/XML feeds).
tidedatabase.py	DbManage class -- all SQLite and InfluxDB read/write operations.
tidealerts.py	TideAlerts class -- evaluates active alert thresholds against incoming readings and triggers email/SMS notifications.
tidenote.py	NotehubReceiver class -- non-blocking HTTP listener for the Blues Notehub webhook (cellular sensor path).
tideprocess.py	Supporting data-processing routines used by the main loop.
tidepredict.py	NOAA predicted-tide retrieval and processing.
tidehtml.py / tidewxhtml.py	Generate the station's tide and weather HTML pages.
tideplot.py	TidePlotRenderer class -- generates the tide/weather plot image and the high/low tide tags; also doubles as tideplot.cgi via a symlink for on-demand rendering.
tidecrypto.py	Centralized key loading and password hashing/verification (Section 7).
tidemonitor.py / tidedisplay.py	Optional local status display (Tkinter-based GUI); not required for core station operation and not used on headless deployments.

5.2 CGI Scripts (Web-Facing)

A small set of CGI scripts, run by Apache under mod_cgid, implement the alert email/SMS subscription workflow: registration and login (alertform.cgi), password change (changealertpw.cgi), forgot-password request and reset (forgotpass.cgi, reset-pw.cgi, reset-pw-1.cgi), registration-link validation (valuser.cgi), and alert configuration management (processalerts.cgi).

> **NOTE:** As of Python 3.13, the standard library's cgi/cgitb modules -- used by all of these scripts -- were removed outright (not merely deprecated). A community-maintained drop-in replacement, the legacy-cgi PyPI package, is required on any node running Python 3.13 or later; see Section 8.3 and Section 11's requirements.txt for detail. This is a documented stopgap, not a permanent fix.

6. Data Storage

6.1 SQLite

A local SQLite3 database (tides.db, path configured via SQL_PATH in tide.env) is the system of record for sensor readings, tide predictions, and user alert-subscription data. SQLite was chosen for its zero-configuration, single-file simplicity, appropriate for a single-writer, embedded-device deployment; the main known operational consideration is that concurrent access from multiple processes (tide.py's main loop versus the web-facing CGI scripts) can produce lock contention, which is why alert-related email requests are queued through a filesystem-based mail spool (Section 9) rather than writing to the database directly from CGI scripts.

Key tables (columns shown are those directly confirmed in code; some tables may have additional columns not exercised by every code path):

Table	Purpose	Key Columns
sensors	Raw sensor readings	dtime, stationid, distance, plus wind/rain/temperature/battery fields
userpass	Alert-subscription user accounts	dtime, emailaddr (Fernet-encrypted), passwd (argon2id hash, or legacy Fernet pending migration), valstat, valkey
useralerts	Per-user alert threshold configuration	email_address, phone_number (both Fernet-encrypted), plus threshold/state fields read by tidealerts.py
iparams	Per-station calibration and configuration	stationid, station1cal/station2cal/station3cal, s1enable/s2enable/s3enable, s1type/s2type/s3type

A periodic copy of the live database is maintained separately (SQL_COPY in tide.env) specifically so ad hoc queries and diagnostics can be run without risking lock contention against tide.py's own frequent access to the live file.

6.2 InfluxDB

Sensor readings and other time-series data are written in parallel to an InfluxDB instance, used for longer-term trend queries and dashboarding (e.g. Grafana) separately from the SQLite-backed live station display. InfluxDB is installed and configured independently of this codebase -- see Section 10.

7. Security and Encryption

7.1 Overview

Three categories of sensitive subscriber data are protected at rest: email addresses, phone numbers, and passwords. Each uses a mechanism appropriate to whether the original value ever needs to be recovered:

Data	Mechanism	Why
Email address	Fernet encryption (reversible)	Must be recovered to actually send alert email
Phone number	Fernet encryption (reversible)	Must be recovered to actually send alert SMS
Password	argon2id hashing (one-way)	Only ever needs verification, never recovery -- hashing means a compromise of stored data cannot yield working passwords

7.2 Key Management

All three Fernet keys (k1 for email, k2 for phone, k3 -- retained only for legacy password verification, see 7.3) plus a fourth key (ku, used elsewhere for encrypted site configuration) are generated by makekeys.py and stored outside the web-served document root, at /home/tide/bin/tidegauge/, with file permissions restricted to owner read/write and group read-only (the tide group, which the web server's www-data account is also a member of). This location and permission scheme replaced an earlier arrangement where these keys lived inside the web server's document root -- reachable, in principle, by any web-server misconfiguration.

Centralized key loading is handled by tidecrypto.py, imported by every CGI script and by the relevant server-side modules (tidealerts.py, tidedatabase.py) that need to decrypt subscriber data. This ensures there is exactly one place that knows where the keys live, rather than each consumer independently constructing a path.

7.3 Password Hashing and Legacy Migration

Passwords are hashed with argon2id (via the argon2-cffi package), the current OWASP-recommended choice for password storage. tidecrypto.py's verify_password() function transparently supports both newly-hashed passwords and legacy Fernet-encrypted passwords from before this scheme was introduced, distinguishing the two by the fixed '$argon2id$' prefix argon2-cffi always produces (a value that can never appear in Fernet's own base64-based output). When a legacy password is successfully verified, the calling script immediately re-hashes and stores the new value -- migrating that user silently, with no forced password reset, the next time they log in.

7.4 Mail Spool (Alert Email Decoupling)

Rather than writing alert-triggered email requests directly to a shared mailspool.sqlite table from web-facing CGI processes (risking lock contention with tide.py's own frequent database access, and requiring those CGI processes to hold broader database credentials than necessary), outbound email requests are written as small JSON files to a filesystem-based mail spool directory. tide.py's main loop picks these up on a fixed schedule, attempts delivery with retry logic, and moves permanently failed requests to a separate failed/ subdirectory with logging.

8. Web Presentation Layer

8.1 Static Content

Three static HTML files (alertlogin.html, forgotpass.html, index.html) serve as the entry points for the alert-subscription workflow and station homepage; the tide and weather displays themselves (tide.html, tideplot.html) are generated dynamically by tidehtml.py/tidewxhtml.py and tideplot.py respectively.

8.2 CGI Scripts

See Section 5.2 for the full list and purpose of each CGI script. All run under Apache's mod_cgid, executing as the www-data user.

8.3 Python 3.13 Compatibility

> **CAUTION:** Python 3.13 removed the cgi and cgitb standard-library modules outright (PEP 594), after deprecating them in 3.11. Every CGI script in this project still imports them directly. The legacy-cgi PyPI package (confirmed working, version 2.6.4, on the tidebuddy test node) provides a compatible drop-in replacement and must be installed in any node's Python virtual environment before these scripts will run on 3.13 or later. This is documented as a known limitation, not a permanent solution -- see Section 12.

8.4 Remote Access to the Site

Where a station's web presentation needs to be reachable from outside its local network without direct port-forwarding, Cloudflare (via a cloudflared tunnel) is used. [Ken: confirm and expand this description -- see the requirements.txt note requesting the exact role of Cloudflare/cloudflared for this deployment.]

8.5 Site-Specific Customization

index.html and alertlogin.html are otherwise identical across every station, deliberately -- only three things vary per site, each supplied as a separate file so a new deployment never requires editing the shared HTML/CSS/JavaScript itself.

webimage.png

The station's own photo, displayed on both index.html and alertlogin.html. Every page that displays it specifies the SAME 8:5 (1.6:1) aspect ratio in its <img> width/height attributes -- alertlogin.html at 480x300, index.html at 560x350. Because every page shares this ratio, the browser only ever needs to scale the image uniformly to fit each page's display size; it never crops or stretches it, and no blank letterboxing appears on either side.

> **CAUTION:** This only holds because every page is deliberately kept at the same 8:5 ratio. If a future page ever specified a different width:height ratio for this image, the browser's default behavior is to stretch/squash the image to fit those exact dimensions, distorting it -- not to crop or letterbox it automatically. Any new page displaying webimage.png must either match the existing 8:5 ratio, or explicitly use CSS object-fit (cover to crop without distortion, contain to letterbox instead) if a different ratio is genuinely needed.

Supply the source file at 960x600 pixels (exactly double the 8:5 display ratio, so it stays sharp on high-DPI/retina screens while scaling down cleanly on standard displays) rather than at the exact display size. If the original photo isn't already 8:5, crop it to that ratio before saving as webimage.png -- don't rely on the browser to do this correctly, since it will distort rather than crop.

webinfo.txt (optional)

A short, plain-text description of the station, displayed in an information block on index.html. This file is entirely optional -- if it is not present at /var/www/html/webinfo.txt, the information block does not appear at all (index.html checks for it via a client-side fetch() on page load and fails silently if the file is missing). A generic, non-

site-specific version of this file is included in the repository; replace it with station-specific text the same way webimage.png is replaced.

STATION_LOCATION (tide.env)

If set, this existing tide.env variable (already used elsewhere for email subject lines and the local monitor panel's title) is also read by a small CGI script, sitename.cgi, and prepended to index.html's title -- for example, 'Belfast Harbor Tide Station Hub' rather than the generic 'Tide Station Hub'. This reuses the single existing STATION_LOCATION value rather than requiring a separate file, so there is only one place to configure a station's display name. If STATION_LOCATION is unset, or sitename.cgi is not installed in cgi-bin, index.html falls back to the generic title with no error.

9. Alert System

9.1 Email Delivery

Alert and status email is sent via Brevo (a transactional email service), configured via EMAIL_SERVICE='brevo' in tide.env and delivered over an SMTP relay connection using smtplib (part of the Python standard library -- no additional package is required for the email transport itself, only for the account/credentials). A fallback path exists for using a different, generic SMTP server if EMAIL_SERVICE is set to any value other than 'brevo'.

9.2 SMS Delivery

SMS alerts (including administrative status messages, see ADMIN_TEL_NBRS) are sent via a Twilio account, using the twilio Python package's REST client (tidehelper.py).

9.3 Alert Evaluation

tidealerts.py's TideAlerts class evaluates each subscriber's configured threshold against incoming tide readings on every main-loop cycle, tracking state across cycles (self.save_alert_list) so that a threshold crossing triggers exactly one notification rather than repeating on every subsequent reading while the condition remains true.

10. External Service and Account Dependencies

The following external accounts and services are required for full station operation. None of these can be provisioned by installing software alone -- each requires an actual account setup step, and none of this is currently automated by install.sh (see Section 11.1).

Service	Required For	Notes
Notecard.io / Notehub.io	Cellular sensor data path (Section 3.2, Path B)	Only required if a station uses a Blues Notecard rather than (or in addition to) LoRa
Brevo	Alert/status email delivery	SMTP credentials configured via the encrypted secrets store, see tidehelper.py
Twilio	Alert/status SMS delivery	Account SID/auth token configured similarly
Cloudflare + cloudflared	External web access	[Ken: confirm exact configuration/role]
InfluxDB	Time-series data storage	Installed and configured as its own service, independent of this codebase

11. Deployment and Installation

11.1 install.sh

> **CAUTION:** install.sh is a starting point, not a complete or currently accurate deployment script. It does not capture Python package dependencies, external account setup, or several other requirements documented in this manual. A full rework is planned; until then, use this manual's Section 10 and requirements.txt (Section 11.2) as the authoritative list of what actually needs to be in place, rather than relying on install.sh alone.

11.2 Python Package Requirements

requirements.txt (repository root) lists all pip-installable Python dependencies, each annotated with which module needs it and why, plus documentation (in comment form, since these aren't pip-installable) of the non-pip infrastructure and account requirements from Section 10. Notably: cryptography (Fernet), argon2-cffi (password hashing), python-dotenv, influxdb-client, pytz, requests, pyserial, wget, feedparser, twilio, and both suntime and suntimes (two separate, unrelated packages both currently in use -- a known duplication, see Section 12).

11.3 Remote Desktop

TigerVNC provides remote desktop access to a station's Raspberry Pi, used both for the operational purposes described in Section 2.4 and for any setup/maintenance task that benefits from a graphical session rather than SSH alone. TigerVNC replaced an earlier RealVNC-based setup to avoid an ongoing paid subscription requirement; a TigerVNC server package must be installed and enabled on the Pi, with a corresponding TigerVNC viewer on the connecting machine.

12. Known Technical Debt

Items documented here are known, understood limitations -- not urgent, but worth tracking so they aren't rediscovered from scratch later.

cgi.FieldStorage() and the Python 3.13 removal

See Section 8.3. legacy-cgi is a documented, functioning stopgap, not a permanent fix. A proper migration would replace cgi.FieldStorage() with urllib.parse.parse_qsl() for GET/HEAD requests and the standard library's email.message module (or the multipart PyPI package) for POST/PUT, per Python's own migration guidance.

Duplicate sun-calculation packages

Two separate, unrelated PyPI packages -- suntime (used in tidehelper.py) and suntimes (used in tideplot.py) -- provide overlapping sunrise/sunset functionality. This is very likely accidental duplication from different points in the project's history, and a good candidate for future consolidation onto a single package.

install.sh incompleteness

See Section 11.1 -- a full rework is planned but not yet scheduled.

Undocumented database schema

No CREATE TABLE statements exist anywhere in the repository; the schema documented in Section 6.1 was inferred from actual INSERT/SELECT usage across the codebase rather than from a formal schema definition. A migrations/schema.sql file capturing this formally would reduce reliance on institutional knowledge.

13. Appendices

Appendix A -- tide.env Reference

Variable	Purpose
STATION_LOCATION / STATION_LATITUDE / STATION_LONGITUDE / STATION_NAME	Station identity and location, used for sun-times and display labeling
NDBC_LOCATION / NDBC_LATITUDE / NDBC_LONGITUDE / NDBC_STATIONS	National Data Buoy Center station(s) used for supplementary weather/wave data
EMAIL_SERVICE	'brevo' or another SMTP provider identifier (Section 9.1)
SQL_PATH / SQL_COPY	Live database path and its periodic-copy path for safe ad hoc querying
HTML_DIRECTORY / CGI_DIRECTORY / CGI_URL	Web server document root, CGI directory, and public CGI URL
NOAA_STATION / NOAA_STATION_NAME	NOAA predicted-tide station used for the predicted-tide overlay
SENSOR_SOURCE	'lora' or 'notehub' -- selects the active sensor data path (Section 3.2)
SERIAL_PORTS / USB0_BAUDRATE	USB serial configuration for the LoRa receive path
WX_SERVICE / WX_OPEN_URL / WX_UND_STATION_ID / WX_LINK_STATION_ID	Weather data source configuration
TIME_ZONE	Station's local time zone
TIDE_URL / TIDE_STATION_URL	Public URLs for the station's tide page
HOME_DIRECTORY	Path to the tide-gauge script/runtime directory (typically /home/tide/bin/tidegauge)
NWS_LOCAL_GRIDPOINTS / NWS_MARINE_GRIDPOINTS / NWS_RADAR	National Weather Service forecast/radar configuration

Appendix B -- Systemd Services

Service	Runs	Purpose
tide	tide.py	Main station process -- sensor ingestion, storage, alerts, web content generation
tidemonitor	tidemonitor.py	Optional local status display (not required on headless deployments)

Appendix C -- Source Repository

All code referenced in this manual is maintained at github.com/cfi1513840/tide-gauge.