CommWatch - Communication Watch Window

Quick Start Guide

Installation

VS Code Extension

1. Download the `.vsix` file from releases
2. Open VS Code
3. Press `Ctrl+Shift+P` (or `Cmd+Shift+P` on Mac)
4. Type "Install from VSIX" and select the file
5. Reload VS Code

Desktop Application

Windows:

bash

```
# Download and run the installer
CommWatch-Setup-0.1.0.exe
```

macOS:

bash

```
# Download and mount the DMG
open CommWatch-0.1.0.dmg
# Drag to Applications folder
```

Linux:

bash

```
# Download and install the AppImage
chmod +x CommWatch-0.1.0.AppImage
./CommWatch-0.1.0.AppImage
```

CLI Tool

bash

```
# Download binary for your platform
# Add to PATH
export PATH=$PATH:/path/to/commwatch/bin

# Verify installation
commwatch --version
```

First Connection - STM32 Nucleo UART

This example shows connecting to an STM32 Nucleo board via USB CDC ACM.

Hardware Setup:

1. Connect STM32 Nucleo board via USB
2. Board appears as virtual COM port (e.g., COM5 on Windows, /dev/ttyACM0 on Linux)

VS Code Extension:

1. Open Command Palette (`Ctrl+Shift+P`)
2. Run "CommWatch: Open"
3. Select device from dropdown (e.g., "STM32 Virtual COM Port")
4. Configure settings:
    - Baud Rate: 115200
    - Data Bits: 8
    - Stop Bits: 1
    - Parity: None
5. Click "Connect"
6. Monitor incoming EFuse frames in real-time

Desktop App:

1. Launch CommWatch
2. File → Connect
3. Select UART device
4. Set baud rate to 115200
5. Click Connect
6. View frames in monitor window

CLI:

bash

```
# Monitor live
commwatch monitor --proto uart --port /dev/ttyACM0 --baud 115200

# Record session
commwatch record --proto uart --port /dev/ttyACM0 --baud 115200 --out session.json --duration 60

# Replay session
commwatch replay --in session.json --proto uart --port /dev/ttyACM0
```

Understanding EFuse Frames

EFuse is a custom protocol included as an example:

Frame Structure:

```
[0xAA] [Type:1] [Length:2] [Payload:N] [CRC16:2] [0xBB]
```

Example - ADC Reading (Type 0x01):

```
AA 01 00 02 12 34 5F A3 BB
|  |  |  |  |  |  |  |  └─ End marker
|  |  |  |  |  |  └──┴──── CRC-16/CCITT-FALSE
|  |  |  |  └──┴─────────── ADC value (0x1234 = 4660)
|  |  └──┴──────────────── Payload length (2 bytes)
|  └─────────────────────── Message type (0x01 = ADC)
└────────────────────────── Start marker
```

Decoded View:

```
Type: ADC Reading (0x01)
Length: 2 bytes
ADC Raw: 4660
Voltage: 2.81V (scaled: 4660/4095 * 3.3V)
CRC: ✓ Valid (0x5FA3)
```

User Guide

Protocols Supported

UART (Universal Asynchronous Receiver-Transmitter)

Configuration Options:

- Baud Rate: 300 to 3,000,000 bps
- Data Bits: 5, 6, 7, or 8
- Stop Bits: 1, 1.5, or 2
- Parity: None, Even, Odd, Mark, Space
- Flow Control: None, XON/XOFF, RTS/CTS, DSR/DTR

Common Use Cases:

- Serial console debugging
- Sensor communication
- Microcontroller programming
- Industrial equipment interfaces

Example Configuration:


json

```
{
 "device": {
  "type": "uart",
  "path": "/dev/ttyUSB0"
 },
 "options": {
  "baudRate": 115200,
  "dataBits": 8,
  "stopBits": 1,
  "parity": "none"
 }
}
```

SPI (Serial Peripheral Interface)

Configuration Options:

- Mode: 0, 1, 2, or 3 (CPOL/CPHA combinations)
- Clock Speed: Up to 50 MHz
- Bit Order: MSB-first or LSB-first
- CS Polarity: Active-low or active-high

Transaction Composer:

javascript

```
[
  { action: "assert_cs", delay: 0 },
  { action: "write", data: [0x03, 0x00, 0x00] },
  { action: "read", length: 16 },
  { action: "deassert_cs", delay: 10 }
]
```

Supported Adapters:

- FT232H USB-to-SPI
- CH347A USB bridge
- Custom socket-SPI implementations

I²C (Inter-Integrated Circuit)

Configuration Options:

- Bus Speed: 100 kHz (Standard), 400 kHz (Fast), 1 MHz (Fast Plus)
- Address Mode: 7-bit or 10-bit
- Slave Address: 0x00 to 0x7F (7-bit)

Common Operations:

```
Write: [Address << 1 | 0, Register, Data...]
Read: [Address << 1 | 1, Length] → [Data...]
Write-then-Read: [Addr|W, Reg] [Addr|R] → [Data...]
```

Bus Scanning: The tool can scan for devices on the I²C bus by testing all valid addresses.

CAN (Controller Area Network)

Configuration Options:

- Bitrate: 125k, 250k, 500k, 1M bps
- CAN-FD: Extended data rates
- Filters: ID-based message filtering
- Listen-Only: Monitoring without ACKs

Frame Format:

```
Standard (11-bit ID):
[ID:11][RTR:1][DLC:4][Data:0-8]

Extended (29-bit ID):
[ID:29][RTR:1][DLC:4][Data:0-8]
```

Supported Interfaces:

- SocketCAN (Linux: can0, vcan0)
- CANalyst-II USB adapter
- PCAN USB adapter
- Custom UDP bridge

Filter Example:


json

```
{
 "filters": [
  { "id": 0x100, "mask": 0x7F0, "extended": false },
  { "id": 0x18FF0000, "mask": 0x1FFFFF00, "extended": true }
 ]
}
```

Ethernet (UDP/TCP/Raw)

Configuration Options:

- Protocol: UDP, TCP, or Raw sockets
- Interface: Network interface selection
- Port: UDP/TCP port number
- Multicast: Join multicast groups
- BPF Filter: Berkeley Packet Filter expressions

UDP Example:


json

```
{
 "device": { "type": "ethernet" },
 "options": {
  "ethProtocol": "udp",
  "ethPort": 5000,
  "ethHost": "192.168.1.100"
 }
}
```

PCAP Capture: When available, the tool can capture packets using libpcap for detailed analysis.

Features

Live Monitor

Display Modes:

- **Hex:** Show raw bytes in hexadecimal
- **ASCII:** Show printable characters
- **Both:** Combined hex + ASCII view

Features:

- Color-coded TX/RX (blue for transmit, green for receive)
- Timestamp precision to microseconds
- Frame length indicators
- Error highlighting
- Auto-scroll with manual override
- Search and filter

Keyboard Shortcuts:

- `Ctrl+F`: Search
- `Ctrl+C`: Copy selected frames
- `Ctrl+A`: Select all
- `Space`: Pause/resume auto-scroll

TX Builder

Manual Entry:

```
Input: AA 01 00 02 12 34 5F A3 BB
Effect: Sends 9 bytes immediately
```

Presets:

json

```
{
 "presets": [
  {
   "name": "ADC Read",
   "data": "AA 01 00 00 5F A3 BB"
  },
  {
   "name": "Status Query",
   "data": "AA 02 00 00 C1 84 BB"
  }
 ]
}
```

Periodic Sending:

- Set interval in milliseconds
- Click "Start Periodic" to begin
- Useful for testing device timeouts

Burst Mode:

- Send multiple frames rapidly
- Configure burst size and interval
- Stress testing and performance validation

Decoder View

Parsed Fields: Shows decoded frame structure with:

- Field names
- Values (with units if applicable)
- Types (uint8, uint16, float, etc.)
- Raw bytes

CRC Verification:

- ✓ Valid: Green badge
- ✗ Invalid: Red badge with expected vs calculated values

Protocol Support:

- EFuse (custom protocol)
- COBS (Consistent Overhead Byte Stuffing)
- SLIP (Serial Line IP)
- Hex (raw hexadecimal)
- ASCII (text)
- Custom decoders via plugins

Filters & Triggers

Filter Types:

1. **Regex Filter:**

```
Pattern: ^AA.*BB$
Action: Show only matching frames
```

2. **Byte Pattern:**

```
Pattern: AA ?? ?? ?? ?? ?? ?? ?? BB
Action: Colorize matches in yellow
```

3. **Field Predicate:**

```
Field: voltage
Condition: > 3.0
Action: Export to separate file
```

Actions:

- Show: Display only matching frames
- Hide: Suppress matching frames
- Colorize: Highlight with custom color
- Export: Save to file
- Respond: Send automated response

Statistics Panel

Real-Time Metrics:

- RX Messages & Bytes
- TX Messages & Bytes
- Error Count
- Message Rate (msg/s)
- Uptime

Performance Monitoring: The tool can handle 10,000+ messages per second without UI lag thanks to:

- Backpressure handling
- Ring buffers
- Worker thread decoding
- Incremental rendering

Logging & Export

Export Formats:

1. **CSV:**

csv

```
Timestamp,Direction,Length,Hex
1698765432000,rx,9,"AA 01 00 02 12 34 5F A3 BB"
1698765432100,tx,7,"AA 02 00 00 C1 84 BB"
```

2. **JSON:**

json

```
{
 "version": "1.0",
 "frames": [
  {
   "id": "frame-0",
   "timestamp": "1698765432000000000",
   "direction": "rx",
   "raw": [170, 1, 0, 2, 18, 52, 95, 163, 187],
   "decoded": { ... }
  }
 ]
}
```

3. **PCAP-NG:** Compatible with Wireshark for advanced analysis.

Session Management:

- Save: Store current configuration and logs
- Load: Restore previous session
- Bookmark: Mark important events
- Annotations: Add notes to frames

Virtual Devices (Simulators)

Each protocol includes a simulator for testing without hardware:

UART Simulator Modes:

- **Loopback:** Echo all sent data
- **Scripted:** Playback predefined sequence
- **Burst:** Generate high-rate traffic
- **Error Inject:** Simulate transmission errors

Example Scripted Simulation:

json

```
{
 "mode": "scripted",
 "script": {
  "events": [
   { "delay": 0, "action": "receive", "data": [0xAA, 0x01, ...] },
   { "delay": 1000, "action": "receive", "data": [0xAA, 0x02, ...] },
   { "delay": 2000, "action": "error", "error": "timeout" }
  ],
  "loop": true
 }
}
```

CAN Simulator: Generates realistic automotive traffic:

- Engine RPM (0x100)
- Vehicle Speed (0x200)
- Coolant Temperature (0x300)
- OBD-II requests/responses (0x7E0-0x7E7)

Ethernet Simulator:

- UDP echo server
- TCP stream generator
- Multicast traffic

Developer Guide

Architecture

Monorepo Structure:

```
commwatch/
├── packages/           # Shared libraries
│   ├── proto-core/     # Core types & utilities
│   ├── transports-*/   # Protocol adapters
│   ├── decoders/       # Protocol decoders
│   └── ui-kit/         # React components
├── apps/
│   ├── vscode-ext/     # VS Code extension
│   ├── desktop/        # Electron app
│   └── cli/            # Command-line tool
└── tools/
    └── simulators/     # Virtual devices
```

Creating Custom Decoders

Step 1: Define Decoder Class

typescript

```ts
import { ProtocolDecoder, DecodedFrame, FrameField, FrameError } from '@commwatch/proto-core';

export class MyCustomDecoder implements ProtocolDecoder {
  id = 'my-protocol';
  name = 'My Custom Protocol';

  decode(raw: Uint8Array): DecodedFrame | null {
    // Parse raw bytes
    if (raw.length < 4) return null;

    const header = raw[0];
    const type = raw[1];
    const length = (raw[2] << 8) | raw[3];
    const payload = raw.slice(4, 4 + length);

    const fields: FrameField[] = [
      {
        name: 'header',
        value: header,
        type: 'uint8',
        raw: raw.slice(0, 1),
        offset: 0,
      },
      {
        name: 'type',
        value: type,
        type: 'uint8',
        raw: raw.slice(1, 2),
        offset: 1,
      },
      {
        name: 'payload',
        value: payload,
        type: 'bytes',
        raw: payload,
        offset: 4,
      },
    ];

    return {
      protocol: this.id,
      fields,
    };
  }
```

```
  encode(fields: FrameField[]): Uint8Array {
    // Build frame from fields
    const header = (fields.find(f => f.name === 'header')?.value as number) || 0;
    const type = (fields.find(f => f.name === 'type')?.value as number) || 0;
    const payload = (fields.find(f => f.name === 'payload')?.value as Uint8Array) || new Uint8Array(0);

    const length = payload.length;
    const frame = new Uint8Array(4 + length);
    frame[0] = header;
    frame[1] = type;
    frame[2] = (length >> 8) & 0xFF;
    frame[3] = length & 0xFF;
    frame.set(payload, 4);

    return frame;
  }

  validate(raw: Uint8Array): FrameError | null {
    if (raw.length < 4) {
      return {
        code: 'FRAME_TOO_SHORT',
        message: 'Frame must be at least 4 bytes',
        severity: 'error',
      };
    }
    return null;
  }
}
```

Step 2: Register Decoder

typescript

```
// In your application startup
import { MyCustomDecoder } from './my-custom-decoder';

const decoder = new MyCustomDecoder();
decoderRegistry.register(decoder);
```

Step 3: Use in Configuration

json

```json
{
  "protocol": {
    "id": "my-protocol",
    "name": "My Custom Protocol",
    "decoder": "my-protocol"
  }
}
```

Creating Custom Transport Adapters

Step 1: Implement TransportAdapter Interface

typescript

```typescript
import { TransportAdapter, DeviceInfo, AdapterHandle, AdapterOpenOptions } from '@commwatch/proto-core';

export class MyCustomAdapter implements TransportAdapter {
  id = 'my-transport';
  name = 'My Custom Transport';
  type = 'my-transport' as const;

  async listDevices(): Promise<DeviceInfo[]> {
    // Discover available devices
    return [
      {
        id: 'my-device-1',
        name: 'My Device 1',
        type: 'my-transport',
        path: '/dev/mydevice0',
      },
    ];
  }

  async open(dev: DeviceInfo, options: AdapterOpenOptions): Promise<AdapterHandle> {
    // Open device and return handle
    return new MyCustomHandle(dev, options);
  }

  supportsSimulation(): boolean {
    return true;
  }

  async createSimulator(config: SimulatorConfig): Promise<AdapterHandle> {
    return new MyCustomSimulator(config);
  }
}
```

Step 2: Implement AdapterHandle

typescript

```
class MyCustomHandle implements AdapterHandle {
  private readCallbacks: Set<(chunk: Uint8Array, meta?: RxMeta) => void> = new Set();
  private stats: AdapterStats = {
    bytesRx: 0,
    bytesTx: 0,
    messagesRx: 0,
    messagesTx: 0,
    errors: 0,
    uptime: 0,
  };

  constructor(private device: DeviceInfo, private options: AdapterOpenOptions) {
    // Initialize device connection
  }

  async write(frame: Uint8Array): Promise<void> {
    // Send data to device
    this.stats.bytesTx += frame.length;
    this.stats.messagesTx++;
  }

  read(cb: (chunk: Uint8Array, meta?: RxMeta) => void): Unsubscribe {
    this.readCallbacks.add(cb);
    return () => this.readCallbacks.delete(cb);
  }

  async setOptions(opts: Partial<AdapterOpenOptions>): Promise<void> {
    Object.assign(this.options, opts);
  }

  async close(): Promise<void> {
    this.readCallbacks.clear();
  }

  async getStats(): Promise<AdapterStats> {
    return { ...this.stats };
  }
}
```

Plugin System

CommWatch supports runtime plugin loading:

Plugin Manifest (plugin.json):

json

```
{
 "name": "my-commwatch-plugin",
 "version": "1.0.0",
 "type": "decoder",
 "entry": "./dist/index.js",
 "exports": {
  "decoder": "MyCustomDecoder"
 }
}
```

Plugin Entry Point:

typescript

```
export class MyCustomDecoder implements ProtocolDecoder {
 // Implementation
}

export function activate(context: any) {
 context.registerDecoder(new MyCustomDecoder());
}
```

Loading Plugins:

typescript

```
import { PluginManager } from '@commwatch/proto-core';

const pluginManager = new PluginManager();
await pluginManager.loadPlugin('./path/to/plugin');
```

Testing

Unit Tests:

typescript

```typescript
import { describe, it, expect } from 'vitest';
import { MyCustomDecoder } from './my-custom-decoder';

describe('MyCustomDecoder', () => {
  const decoder = new MyCustomDecoder();

  it('should decode valid frame', () => {
    const frame = new Uint8Array([0x55, 0x01, 0x00, 0x04, 0x12, 0x34, 0x56, 0x78]);
    const decoded = decoder.decode(frame);

    expect(decoded).not.toBeNull();
    expect(decoded?.fields[0].value).toBe(0x55);
  });

  it('should detect invalid frame', () => {
    const frame = new Uint8Array([0x55, 0x01]);
    const error = decoder.validate(frame);

    expect(error).not.toBeNull();
    expect(error?.code).toBe('FRAME_TOO_SHORT');
  });
});
```

Integration Tests with Simulators:

typescript

```
import { describe, it, expect } from 'vitest';
import { UARTAdapter } from '@commwatch/transports-uart';

describe('UART Integration', () => {
  it('should send and receive data', async () => {
    const adapter = new UARTAdapter();
    const handle = await adapter.createSimulator({ mode: 'loopback' });

    const received: Uint8Array[] = [];
    handle.read((chunk) => {
      received.push(chunk);
    });

    const testData = new Uint8Array([0x01, 0x02, 0x03]);
    await handle.write(testData);

    await new Promise(resolve => setTimeout(resolve, 100));

    expect(received.length).toBeGreaterThan(0);
    expect(Array.from(received[0])).toEqual([0x01, 0x02, 0x03]);

    await handle.close();
  });
});
```

Building and Packaging

Build All Packages:

bash

```
pnpm install
pnpm run build
```

Package VS Code Extension:

bash

```
cd apps/vscode-ext
pnpm run package
# Output: commwatch-vscode-0.1.0.vsix
```

Build Electron App:


bash

```
cd apps/desktop

# Build for current platform
pnpm run dist

# Build for specific platforms
pnpm run dist:win
pnpm run dist:mac
pnpm run dist:linux

# Output: apps/desktop/release/
```

Build CLI Binary:


bash

```
cd apps/cli
pnpm run build:binary

# Output: apps/cli/bin/
# - commwatch-win-x64.exe
# - commwatch-linux-x64
# - commwatch-macos-x64
```

Performance Optimization

High-Rate Data Handling:

1. **Backpressure Management:**


typescript

```
const RING_BUFFER_SIZE = 10000;
const ringBuffer = new RingBuffer<ProtocolFrame>(RING_BUFFER_SIZE);

handle.read((chunk) => {
  if (!ringBuffer.isFull()) {
    ringBuffer.push(processFrame(chunk));
  } else {
    // Drop frame or apply other strategy
    stats.errors++;
  }
});
```

2. **Worker Thread Decoding:**

typescript

```
const decoderWorker = new Worker('./decoder-worker.js');

decoderWorker.postMessage({ raw: chunk });
decoderWorker.onmessage = (e) => {
  const decoded = e.data;
  updateUI(decoded);
};
```

3. **Incremental Rendering:**

typescript

```
// Render only visible frames
const virtualScroll = useVirtualScroll({
  itemCount: frames.length,
  itemHeight: 24,
  windowHeight: 600,
});

return frames.slice(virtualScroll.startIndex, virtualScroll.endIndex).map(renderFrame);
```

Debugging

Enable Debug Logging:

bash

```
# VS Code Extension
Set environment variable: DEBUG=commwatch:*

# Desktop App
Launch with --enable-logging flag

# CLI
Use --verbose flag
```

Log Levels:

- ERROR: Critical failures
- WARN: Potential issues
- INFO: General information
- DEBUG: Detailed diagnostics
- TRACE: Very detailed traces

Common Issues:

1. **Permission Denied on Serial Port (Linux):**

bash

```
sudo usermod -a -G dialout $USER
# Logout and login again
```

2. **CAN Interface Not Found:**

bash

```
# Load virtual CAN module
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
```

3. **High CPU Usage:**

- Reduce frame rate with filters
- Disable real-time decoding for high-rate protocols
- Use ring buffer with size limits

Examples & Sample Configurations

STM32 Nucleo UART Configuration

json

```
{
 "name": "STM32 Nucleo UART",
 "device": {
  "id": "uart:COM5",
  "name": "STM32 Virtual COM Port",
  "type": "uart",
  "path": "COM5"
 },
 "protocol": {
  "id": "efuse",
  "name": "EFuse Protocol",
  "decoder": "efuse"
 },
 "options": {
  "baudRate": 115200,
  "dataBits": 8,
  "stopBits": 1,
  "parity": "none"
 },
 "presets": [
  {
   "name": "Read ADC",
   "data": "AA 01 00 00 5F A3 BB"
  },
  {
   "name": "Read Status",
   "data": "AA 02 00 00 C1 84 BB"
  },
  {
   "name": "Write Config",
   "data": "AA 03 00 04 12 34 56 78 E2 F1 BB"
  }
 ]
}
```

CAN Bus OBD-II Configuration

json

```
{
 "name": "OBD-II CAN",
 "device": {
  "type": "can",
  "path": "can0"
 },
 "options": {
  "canBitrate": 500000,
  "canFilters": [
   { "id": 0x7E8, "mask": 0x7FF, "extended": false },
   { "id": 0x7E9, "mask": 0x7FF, "extended": false }
  ]
 },
 "presets": [
  {
   "name": "Request Engine RPM",
   "data": "00 00 07 E0 02 01 0C 00 00 00 00 00"
  },
  {
   "name": "Request Vehicle Speed",
   "data": "00 00 07 E0 02 01 0D 00 00 00 00 00"
  }
 ]
}
```

Ethernet UDP Configuration

json

```
{
 "name": "UDP Communication",
 "device": {
  "type": "ethernet"
 },
 "options": {
  "ethProtocol": "udp",
  "ethPort": 5000,
  "ethHost": "192.168.1.100"
 },
 "filters": [
  {
   "type": "pattern",
   "value": "STAT",
   "action": "colorize",
   "color": "green"
  },
  {
   "type": "pattern",
   "value": "ERR",
   "action": "colorize",
   "color": "red"
  }
 ]
}
```

FAQ

Q: Can I use CommWatch without physical hardware? A: Yes! Each protocol includes a simulator. Use the simulator option when connecting.

Q: How do I add support for a custom protocol? A: Create a custom decoder implementing the `ProtocolDecoder` interface. See "Creating Custom Decoders" section.

Q: What's the maximum data rate supported? A: CommWatch can handle 10,000+ messages/second with proper configuration. Use ring buffers and worker threads for optimal performance.

Q: Can I export captured data to Wireshark? A: Yes, use the PCAP-NG export format which is compatible with Wireshark.

Q: Does it work on Raspberry Pi? A: Yes, the CLI and desktop app work on Raspberry Pi (ARM Linux). Build from source or use the Linux ARM binaries.

Q: How do I report bugs or request features? A: Open an issue on GitHub with detailed description and steps to reproduce.

Q: Is there a limit to log file size? A: The tool automatically rotates logs when they reach configurable size limits (default 100MB).

Q: Can I script automated tests? A: Yes, use the CLI tool with replay functionality or the built-in JavaScript scripting engine.

Troubleshooting

Device Not Found

Symptoms: Device doesn't appear in device list

Solutions:

1. Check physical connection
2. Verify drivers are installed
3. Check permissions (Linux/macOS)
4. Try different USB port
5. Refresh device list

Connection Fails

Symptoms: "Failed to open device" error

Solutions:

1. Close other applications using the device
2. Check device is not in use
3. Verify configuration (baud rate, etc.)
4. Try simulator mode to isolate hardware issues

High CPU Usage

Symptoms: Application becomes slow or unresponsive

Solutions:

1. Apply filters to reduce displayed frames
2. Increase frame buffer size
3. Disable real-time decoding
4. Export and analyze offline

CRC Errors

Symptoms: Many frames show CRC mismatch

Solutions:

1. Check cable quality and connections
2. Reduce baud rate (UART)
3. Verify protocol configuration
4. Check for EMI/noise issues

License & Credits

CommWatch is open source software.

Dependencies:

- serialport: Serial port communication
- socketcan: CAN bus support (Linux)

- electron: Desktop application framework
- VS Code Extension API
- React & Tailwind CSS

Special Thanks:

- STM32 community for testing
- CAN bus protocol contributors
- All open source dependencies

Support

- Documentation: https://docs.commwatch.io
- GitHub: https://github.com/commwatch/commwatch
- Issues: https://github.com/commwatch/commwatch/issues
- Discord: https://discord.gg/commwatch