Technical Details: Engineering Practices from Side Project

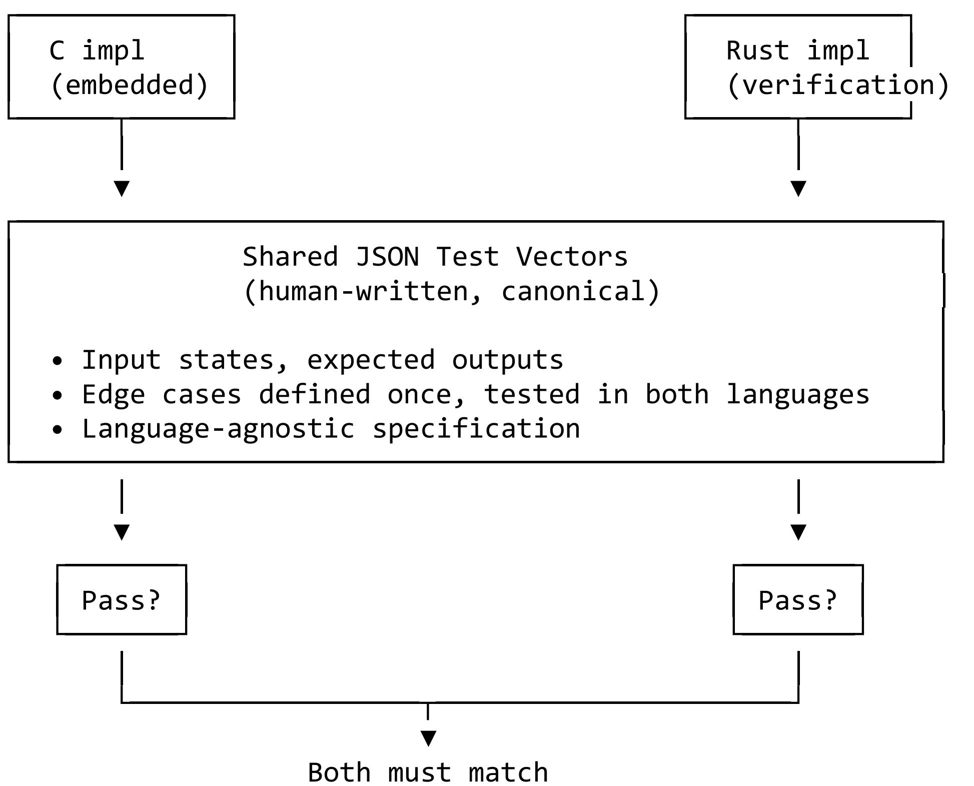
Supplementary material for email to NCR Voyix CTO Author: Bojan Janjatovic, Senior Software Engineer, Platform Engineering **Date:** January 2026

Table of Contents

---

1. Dual-Implementation Verification

Architecture



Test Vector Example

```json
{
  "test": "leader_election_timeout",
  "initial_state": {
    "role": "follower",
    "term": 5,
    "votes_received": 0,
    "election_timeout_ms": 150
  },
  "input": {
    "event": "timeout_elapsed",
    "elapsed_ms": 200
  },
  "expected_output": {
    "role": "candidate",
    "term": 6,
    "voted_for": "self",
    "action": "broadcast_vote_request"
  }
}
```

Why This Works

- **Different failure modes**: C bugs (buffer overflows, pointer errors) differ from Rust bugs (logic errors, type mismatches)
- **AI cross-check**: If AI generates C code with a subtle bug, it's unlikely to generate the identical bug in Rust
- **Specification as code**: JSON vectors serve as executable documentation

---

2. Agentic AI Development - Two Modes

Not all tasks are equal. I distinguish between two fundamentally different modes of AI-assisted development:

Mode A: Extreme Agentic (Autonomous)

```
EXTREME AGENTIC MODE
─────────────────────────────────────────────────────────────

When to use:
  • Well-defined implementation task
  • Clear success criteria exist (test vectors, specs)
  • Correctness is automatically verifiable

How it works:
  1. Human defines task and success criteria
  2. AI works autonomously for 1+ hours
  3. No human input during session
  4. AI commits to git, runs tests, iterates
```

```
  5. Human reviews RESULT, not every step

Verification:
  • Test vectors pass in both C and Rust → implementation correct
  • Emulation tests pass → behavior correct
  • Human reviews final diff, not intermediate steps

Example tasks:
  • "Implement heartbeat protocol per this spec"
  • "Add thermal derating per this formula"
  • "Parse CAN message ID 0x300 per this format"
```

Mode B: Human-in-the-Loop (Guided)

```
HUMAN-IN-THE-LOOP MODE
```

```
When to use:
  • Exploratory work, research
  • Architecture decisions
  • No clear test criteria
  • Correctness cannot be automatically verified

How it works:
  1. Human and AI collaborate step-by-step
  2. Human reviews each proposed change
  3. Explicit permission grants per action
  4. Slower, but necessary for ambiguous tasks

Example tasks:
  • "Help me design the state machine"
  • "What's the best approach for X?"
  • "Refactor this module for clarity"
  • "Write documentation for this feature"
```

Decision Matrix

```
┌──────────────────────────────────────────────────────────────┐
│                                                              │
│   Can correctness be          YES → EXTREME AGENTIC          │
│   automatically verified?           (autonomous, fast)       │
│   (test vectors, specs)                                      │
│                               NO  → HUMAN-IN-THE-LOOP        │
│                                     (guided, careful)        │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

Key Insight

Test vectors are the enabler of autonomous AI development. Without them, human oversight is mandatory. The dual-implementation approach (C + Rust + shared vectors) creates the verification infrastructure that makes extreme agentic mode safe.

```
TEST VECTORS → AUTONOMOUS AI POSSIBLE
NO TEST VECTORS → HUMAN MUST GUIDE
```

This matches how we think about other automation: automated testing enables continuous deployment; without tests, manual QA is required.

---

3. Go for Backend Services

No Magic Programming

```go
// Go: Explicit error handling - no hidden control flow
func (s *Simulation) AddModule(id uint8) error {
    if id >= MaxModules {
        return fmt.Errorf("module ID %d exceeds max %d", id, MaxModules)
    }
    if _, exists := s.modules[id]; exists {
        return ErrModuleAlreadyExists
    }
    // ...
    return nil
}
```

What "no magic" means in practice: - No exceptions flying up the stack invisibly - No inheritance hierarchies to trace - No decorators, annotations, or runtime reflection surprises - No dependency injection frameworks with hidden wiring - Code reads top-to-bottom, left-to-right

When debugging at 3 AM, "boring" code is a feature.

Docker / Container Benefits

```dockerfile
# Go Dockerfile - entire thing
FROM scratch
COPY myservice /myservice
ENTRYPOINT ["/myservice"]
```

Container Image Size Comparison

Technology	Image Size	Notes
Java (Spring Boot)	400-800 MB	JVM + dependencies
Python (Flask)	150-400 MB	Interpreter + packages
Node.js (Express)	150-300 MB	Runtime + node_modules

Technology	Image Size	Notes
Go	**10-20 MB**	Static binary, no runtime

Why this matters: - Faster deployments (10 MB pulls faster than 400 MB) - Smaller attack surface (no runtime = fewer CVEs) - FROM scratch possible (empty base image) - No runtime dependencies - Millisecond startup (no JVM warmup)

Go Benefits Summary

Aspect	Go Advantage
Error handling	Explicit, in-band, no hidden flow
Inheritance	None - composition only
Dependencies	Single binary, zero runtime deps
Containers	10 MB images, FROM scratch
Startup	Milliseconds
Concurrency	Built-in race detector
Learning curve	Small language, readable in days

---

4. Rust for Security-Critical Parsers

Code Example

```rust
// Rust: Compiler enforces handling all cases
enum ModuleState {
    Offline,
    Initializing { start_time: Instant },
    Online { neighbor_count: u8 },
    Fault { code: FaultCode, recoverable: bool },
}

match module.state {
    ModuleState::Offline => self.start_discovery(),
    ModuleState::Fault { recoverable: true, .. } => self.attempt_recovery(),
    ModuleState::Fault { recoverable: false, .. } => self.halt(),
    // Compiler error if cases not exhaustive
}
```

Rust Benefits

- No null pointer dereferences (Option instead)
- No data races (borrow checker enforces at compile time)
- Pattern matching forces handling all cases
- Memory safety without garbage collection

---

5. Distributed Consensus for Edge Devices

Architecture Comparison

```
TRADITIONAL                       DISTRIBUTED
──────────────────────────────────────────────────────────────
┌─────────────┐                   ┌─────────────┐
│  Central    │                   │  Device A   │◄───┐
│  Controller │                   └─────────────┘    │
└─────────────┘                                      │  peer-to-peer
       │                          ┌─────────────┐    │
       ▼                          │  Device B   │◄───┼──── consensus
                                  └─────────────┘    │
┌──────────────┐                                     │
│ All devices  │                  ┌─────────────┐    │
│ depend on it │                  │  Device C   │◄───┘
└──────────────┘                  └─────────────┘

                                  Any device can lead.
Single point of failure.          Others continue if one fails.
```

Key Insight

The same Raft consensus used in etcd/Consul works for coordinating physical devices, not just database replicas.

---

6. Emulation-Based Testing

Renode Architecture

```
┌──────────────────────────────────────────────────────────────┐
│                      RENODE EMULATOR                          │
│  ┌─────────────┐   ┌─────────────┐   ┌─────────────┐         │
│  │ STM32G474   │ ─ │  CAN-FD     │ ─ │  Simulated  │         │
│  │ (emulated)  │   │  Bus        │   │  Neighbors  │         │
│  │             │   │  (virtual)  │   │             │         │
│  └─────────────┘   └─────────────┘   └─────────────┘         │
│                                                               │
│  Runs ACTUAL firmware binary, not simulation                  │
└──────────────────────────────────────────────────────────────┘
```

Benefits

- Test firmware without hardware
- Reproducible CI/CD pipeline
- Catch “works on my machine” issues
- Test failure scenarios safely

---

7. Symbolic AI for Deterministic Systems

LLM vs Safety-Critical Requirements

```
LLM CHARACTERISTICS
─────────────────────────────────────────────────────────────
✗ Non-deterministic    Same input can produce different output
✗ No guarantees        "Usually correct" ≠ "always correct"
✗ Black box            Cannot prove why a decision was made
✗ Hallucinations       Confidently wrong outputs
✗ No formal verification possible

REQUIRED FOR SAFETY-CRITICAL SYSTEMS
─────────────────────────────────────────────────────────────
✓ Deterministic        Same input → same output, always
✓ Provable             Can formally verify correctness
✓ Explainable          Clear audit trail of decisions
✓ Bounded behavior     Known limits, predictable edge cases
✓ Certifiable          Meets ISO 26262, IEC 61508, etc.
```

Symbolic AI Approach

```
RULE ENGINE EXAMPLE
─────────────────────────────────────────────────────────────
IF voltage > 920V AND current > 0
  THEN fault(OVERVOLTAGE), action(DISCONNECT)

IF temperature > 85°C AND power > 2.5kW
  THEN action(DERATE), set_power(2.0kW)

IF heartbeat_timeout > 500ms
  THEN state(SUSPECT), action(PROBE)
─────────────────────────────────────────────────────────────

Results:
• DETERMINISTIC   - Same conditions → same actions
• EXPLAINABLE     - "Why did it disconnect?" → logged rule
• TESTABLE        - Every rule has explicit test cases
• CERTIFIABLE     - Formal methods can prove properties
```

Automotive Certification Example (ISO 26262)

Requirement: Demonstrate that software behaves correctly under ALL conditions

- **LLM approach:** “It worked in 99.7% of test cases” → NOT CERTIFIABLE
- **Symbolic approach:** “Here is formal proof that condition X always produces action Y” → CERTIFIABLE

Hybrid Approach: LLM + Symbolic

```
┌──────────────────────────────────────────────────────────────┐
│                                                              │
│   LLM (Claude, GPT)              SYMBOLIC LAYER              │
│   ─────────────────              ─────────────────           │
│   • Development assistance       • Runtime decisions         │
│   • Code generation              • Safety interlocks         │
│   • Documentation                • Fault handling            │
│   • Test case ideation           • State machine logic       │
│                                                              │
│   Used at DESIGN TIME            Used at RUN TIME            │
│   (human in the loop)            (deterministic execution)   │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

My project uses both: - LLM (Claude) helps write code, explore designs, generate documentation - Symbolic logic (state machines, explicit rules) runs on the actual device - The LLM never runs in production - only deterministic code does

Expert Systems Renaissance

Era	Approach	Limitation
1980s	Expert systems	Knowledge acquisition bottleneck
2020s	LLMs	Non-deterministic, can't certify
Now	Hybrid	LLM helps BUILD expert systems, symbolic logic RUNS them

---

Contact

Bojan Janjatovic Senior Software Engineer, Platform Engineering NCR Voyix

Available for deeper discussion on any of these topics.