AI Coding:
From Research to Millions of Developers

Martin Monperrus

Professor of Software Technology
KTH Royal Institute of Technology, Stockholm

IEEE Fellow
"for pioneering the use of machine learning in
code assistance and program repair,
impacting millions of developers"

IEEE Sweden General Assembly, 2026

Today's Journey

1. What is AI Coding?
2. Research Highlights
3. The Field Today and Tomorrow

Software is Eating the World

- Every modern system runs on code: aircraft, power grids, medical devices, financial markets, phones
- $\approx$ 30 million professional software developers worldwide (2024)
- Writing code is **hard, slow, and error-prone**
- A single bug can cost millions — or lives

The central tension

Demand for software grows faster than the supply of developers.

Can machines help write code?

What Does a Developer Actually Do?

1. **Understand** a problem or specification
2. **Write** code that solves it
3. **Test** whether the code is correct
4. **Debug** / **repair** when tests fail
5. **Maintain** and evolve code over years

AI Coding = automating or assisting any of these steps

Today we focus mostly on **writing** and **repairing** code.

The Key Insight: Code Is Language

Natural language (English):

```
The cat sat on the mat.
```

Programming language (Python):

```python
def greet(name):
    return "Hello, " + name
```

Fundamental observation (ca. 2009–2012)

Source code has **statistical regularities** just like natural language.

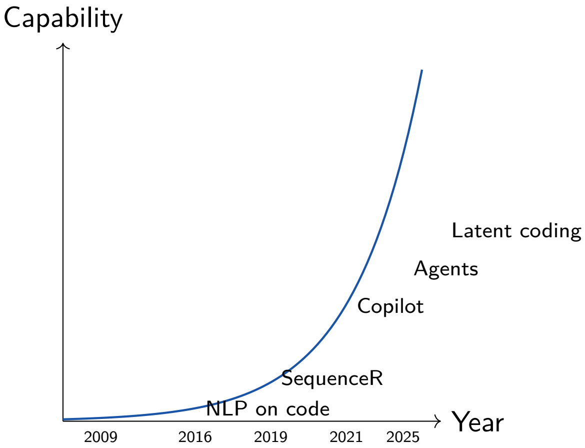
Sequences of tokens that appear often are likely to appear again — and can be *predicted*.

Hindle et al., *On the Naturalness of Software*, ICSE 2012 (Devanbu)

A Brief History of AI Coding (I)

- **1950s** First compilers: machines translate high-level text to machine code
- **1970s–90s** Syntax checkers, early static analysis
- **2009** **First training AI on code** (FSE 2009)
- **2012** *"On the Naturalness of Software"* — code modeled as language
- **2019** **Repairnator** — robot submits accepted patches to real projects; **SequenceR** — seq2seq end-to-end neural repair *(Monperrus et al.)*
- **2021** **GitHub Copilot** — LLMs for code completion, goes mainstream
- **2022** **Execution-based backpropagation** — test outcomes as training signal *(Ye, Martinez, Monperrus)*; neural repair of **security vulnerabilities** in C *(Chen, Kommrusch, Monperrus)*
- **2024–26** GPT, Claude, Gemini: conversational coding agents;

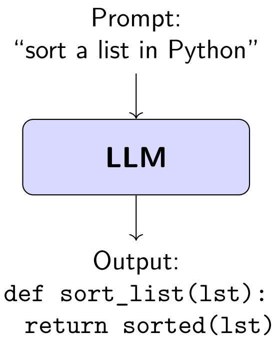
A Brief History of AI Coding (II): The Exponential



How Modern AI Coding Works — The Big Picture

Large Language Model (LLM)

- Trained on billions of lines of code + text
- Learns statistical patterns of tokens
- Given a *prompt*, predicts the next most likely tokens
- No explicit rules — emergent competence



Key point for engineers from other fields

This is **pattern completion at massive scale**—
the results are increasingly useful and sometimes remarkable.

Three Flavors of AI Coding

1. Code Completion

Developer types; AI continues.

Example: GitHub Copilot

Used by **millions daily**

2. Code Generation

Developer describes in English; AI writes code from scratch.

Example: ChatGPT, Claude

Changes **who can code**

3. Program Repair

AI finds a bug and fixes it automatically.

My research focus

Hardest problem — requires **understanding correctness**

*These three are converging into unified **AI coding agents**.*

Why Program Repair Is Hard

To fix a bug automatically, a system must:

1. Know what *correct behavior* looks like
2. Find the *location* of the fault
3. Generate a *valid patch*
4. Verify the patch *does not break anything else*

Each step is a research problem in itself.

The oracle problem

How do you know code is *correct*?

Test suites are the practical proxy: a patch passes if all tests pass.

But tests are incomplete — and this shapes everything.

My Research Arc (2009 – 2026)

- Open-source tools and datasets used worldwide

Paper #1 — Learning from Examples (FSE 2009)

"Learning from Examples to Improve Code Completion Systems"

ESEC/FSE 2009 — Bruch, Monperrus & Mezini

ACM SIGSOFT Impact Paper Award

Idea: Instead of syntactic completion rules, *learn* which API calls co-occur from a large corpus of existing programs.

- Mine usage patterns from thousands of open-source projects
- Rank completion suggestions by learned probability
- Works for any object-oriented API

Result

Dramatic improvement in completion accuracy over Eclipse's built-in completions on real Java APIs.

Foundational insight: code repositories are a goldmine of statistical knowledge about how code is written.

Paper #1 — Why It Started Everything

- **First paper** to show that *machine learning on large code corpora* directly improves developer tools
- Established the paradigm: **mine code** → **learn patterns** → **assist developers**
- Predates Hindle et al. (2012) and the "naturalness" wave — an independent but parallel insight
- Awarded the **ACM SIGSOFT Impact Paper Award**
- Every modern AI coding tool (Copilot, Cursor, . . .) is a scaled-up version of this core idea: learn from code history what comes next

Paper #2 — Repairnator (2019)

"Human-competitive Patches in Automatic Program Repair with Repairnator"

ICSE 2019 / Software Engineering in Practice

Idea: Build a software *robot* that:

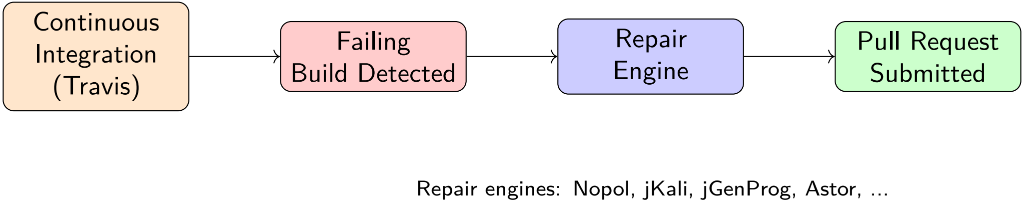
1. Monitors CI builds on GitHub continuously
2. Detects failing builds (= bugs caught by tests)
3. Applies automated repair techniques
4. Submits pull requests with patches

Result

Repairnator generated patches that were **accepted by human developers** in real open-source projects — indistinguishable from human patches.

First time ever.

Paper #2 — Repairnator: Architecture



- Ran 24/7 on 14,000+ real CI builds per month
- **Key insight:** automation enables *scale* — no human could monitor this
- Showed that repair is not just an academic exercise, shower human-competitiveness

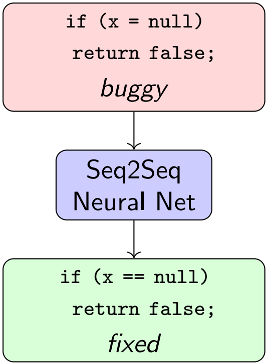
Paper #3 — SequenceR (2019)

“SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair”

IEEE Transactions on Software Engineering, 2019
(Chen, Kommrusch, Tufano, Pouchet, Poshyvanyk, Monperrus)

Idea: Treat bug repair as a *translation* task.

- Input: buggy code line
- Output: fixed code line
- Model: sequence-to-sequence (encoder-decoder) neural network, trained on thousands of past bug fixes



Paper #3 — SequenceR: Why It Matters

- **First** end-to-end neural network for program repair
- Introduced the idea of *copy mechanism*: the model can copy tokens from the surrounding context (crucial for code), still today in coding agents
- No hand-crafted rules — **learned entirely from data**
- Published in IEEE TSE 2019 — one of the most-cited works in neural repair

What changed

Before SequenceR: repair needed hand-coded templates and domain knowledge.
After SequenceR: a neural model trained on commit history could repair code.
This was the **transition point** to modern AI-driven program repair.

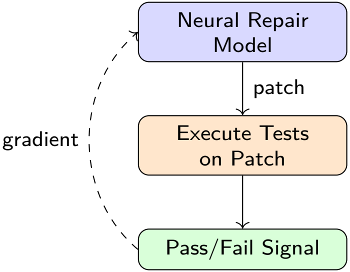
Paper #4 — Neural Repair with Execution (ICSE 2022)

"Neural Program Repair with Execution-based Backpropagation"

ICSE 2022 — He Ye, Matias Martinez, Martin Monperrus

Problem: Previous neural repair models are trained only on *syntactic* correctness (does it look like a a patch?).

Idea: Train the model using *test execution feedback* — run the patch, see if tests pass, backpropagate that signal.



Paper #4 — Why Execution Feedback Is a Breakthrough

- The model learns that *semantic correctness* (tests passing) is the goal, not just syntactic plausibility
- Bridges the gap between **statistical language models** and **program semantics**
- Analogous to reinforcement learning from human feedback (RLHF) — but the "human feedback" is replaced by **automated test execution**
- Significant improvement on Defects4J benchmark

Broader significance

This paper laid groundwork for using automated oracles (tests, compilers, formal verifiers) as training signals — a paradigm now central to the latest code LLMs and coding agents.

Paper #5 — Security Vulnerability Repair (TSE 2022)

"Neural Transfer Learning for Repairing Security Vulnerabilities in C Code"

IEEE Transactions on Software Engineering, 2022
(Chen, Kommrusch, Monperrus)

IEEE TSE Best Paper Award

Motivation: Security vulnerabilities (buffer overflows,
SQL injections, ...) are catastrophic. Manual patching is slow.

Idea: Transfer a model trained on general bug fixes to the domain of *security vulnerabilities* in C.

Impact

- Demonstrated that neural repair generalizes across domains
- Security patching is a billion-dollar problem — automation matters
- Used CVE datasets for evaluation

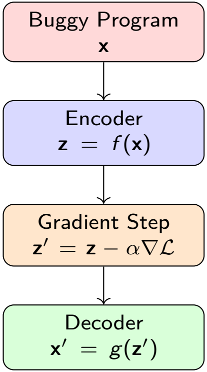
Paper #6 — Gradient-Based Program Repair

"Repairing Programs Directly in Continuous Latent Space"

Most futuristic direction, current research in my group

Key idea: Instead of generating a patch token-by-token, *differentiate through the program representation* directly.

- Encode the buggy program into a continuous vector
- Use gradient descent to move the vector toward a correct program
- Decode back into source code
- No discrete search — fully differentiable pipeline



Paper #6 — Why Gradient-Based Repair Matters

- **Escapes the combinatorial explosion** of discrete patch search: gradient descent is orders of magnitude more directed
- **Unified training signal**: the correctness loss (test pass/fail) flows directly back in the vector space
- **Connects program repair to the topology of code embeddings**: bugs are points in latent space; fixes are nearby, reachable by gradient steps
- Opens the door to **simultaneous multi-location repair** as a single optimisation problem

Long-term vision

If programs can be continuously optimised like neural network weights, the boundary between *training a model* and *repairing software* dissolves, a fundamental shift in how we think about correctness.

Common Thread Across Papers

Paper	**Key Contribution**	**Legacy**
Learning from Ex. (FSE'09)	ML on code corpora	Copilot paradigm
Repairnator (2019)	Repair at CI scale	Industrial deployment
SequenceR (2019)	End-to-end neural repair	Coding agent
Exec. BP (2022)	Semantic training signal	Post-training RL
Security (TSE'22)	Transfer to security	Practical impact
Gradient repair	Differentiable repair	Future paradigm

Each paper pushed one boundary; together they define a research program.

What AI Coding Tools Can Do Today

Impressive successes:

- Repair: Fix **complex, multi-file bugs** spanning thousands of lines
- **Write entire programs and almost systems** from a natural-language description
- Complete features end-to-end: spec → code → tests → docs
- Translate and modernize legacy codebases (COBOL, Fortran)
- Solve competitive programming problems (Olympiad level)
- Pass senior-level coding interviews

Persistent limitations:

- Hallucinate non-existent APIs
- Miss subtle semantic and concurrency errors
- Cannot reliably verify own output
- Poor at long-horizon architectural integrity
- Security vulnerabilities introduced silently

Open Research Problems I Find Most Exciting

1. **Test-time compute for repair**
   Let the model "think longer" on hard bugs — analogous to AlphaGo's MCTS
2. **Learning from execution at scale**
   Use compilers, fuzzers, formal verifiers and tests as infinite training signal
3. **Latent-space program synthesis**
   Operate directly in embedding space, not token-by-token — potentially orders of magnitude faster
4. **ARC-AGI and beyond**
   Program synthesis as a path to general intelligence

The Human-AI Collaboration Question

Not replacement — augmentation

- Developers using Copilot are -20%, 100% faster on benchmark tasks
- But: code review load *increases* — more code to read
- Skills that become more valuable: system design, requirements, verification
- Skills that become less valuable: boilerplate, syntax memorization

Key problems

Reliability?
Liability?
Economics?

Reflections: What I Got Right and Wrong

Predictions that held:

- Code *is* a language (most) amenable to ML
- Neural approaches *would* dominate repair
- Execution feedback *is* crucial for training
- Scale matters — big models beat clever heuristics

What I underestimated:

- **Speed** of LLM capability jumps
- **Breadth**: LLMs generalize across dozens of programming languages with no fine-tuning
- **Adoption**: millions of users in months, not years
- **Societal impact**: education, workforce — all disrupted simultaneously

For Engineers from Other Fields: The Takeaway

AI Coding idemonstrably works at scale

- Code is *formal* enough that correctness is verifiable (unlike text or images)
- Code has *abundant training data* (billions of lines on GitHub)
- The feedback loop is *fast and cheap* (just run the tests)

This makes AI coding a *leading indicator*:
the techniques developed here — neural search, execution-based training, iterative agentic loops — are being applied to:

- Mathematical proof
- Hardware, chip design (VHDL, SystemVerilog, Google's AlphaChip)
- AI for science

Summary

1. **AI Coding** = using machine learning to write and maintain code
2. The key insight: **code is the most learnable language** — statistical regularities and clear semantics enable prediction
3. My research arc: from code completion (2009)
   to SequenceR's neural repair (2019) to execution-driven training (2022)
   to large-language models (2023)
4. **Today**: millions of developers use AI daily; entire startups built on this
5. **Tomorrow**: agentic systems that verify their own output, learn from execution, and eventually produce reliable systems.

Acknowledgments

Collaborators & students (selected):
He Ye, Matias Martinez, Zimin Chen,
Steve Kommrusch, Deepika Tiwari,
Benoit Baudry, and many more at
KTH, University of Lille, Virginia Tech

Funding:
Wallenberg AI, Autonomous Systems and
Software Program (WASP),
Swedish Foundation for Strategic Research (SSF)

The open-source community —
without whom none of this is possible.

A personal note

When I started working on
automated program repair in 2011,
it was considered an impossible curiosity.

It has become a multi-billion $ industry.

Thank you

AI Coding: From Research to Millions of Developers

`monperrus@kth.se`
`https://www.monperrus.net/martin/publications`

Papers, datasets, and tools: all open-source.
The living review on program repair is continuously updated at
`monperrus.net/martin/repair-living-review`

Slides made with LaTeX Beamer.