Principal Investigator: Martin Monperrus
Host Institution: KTH Royal Institute of Technology

Software Engineering in the Latent Space

Acronym: LATENT

Duration: 60 months

Abstract (max 2000 characters):

Programs are symbolic objects: discrete, structured, executable, but opaque to numerical optimisation. Large language models have created a parallel world in which programs, bugs, patches, tests, and specifications exist implicitly as vectors in a continuous latent space. These two worlds are not yet connected as a scientific discipline. LATENT creates that discipline: software engineering in the latent space.

The central hypothesis is that a duality holds between symbolic programs and their continuous latent representations, and that this latent geometry faithfully captures software semantics. If it does, problems that are undecidable or intractable in the symbolic world become approachable through geometric and gradient-based operations on vectors.

LATENT tests this hypothesis along three objectives. Representation: discover which properties of programs, tests, specifications, and architectures are encoded in learned latent spaces, and whether they stay stable across models, languages, and semantics-preserving transformations. Intervention: determine when latent representations can be causally manipulated to generate, repair, and secure code, separating features that merely correlate with a property from those that control model behaviour. Verification: bridge continuous search back to symbolic guarantees, turning latent trajectories into programs, tests, or counterexamples whose correctness is checked by executable oracles.

Program repair, semantic equivalence, specification enforcement, and architectural integrity serve as demanding semantic testbeds. The project will release open benchmarks and, more novel, datasets of latent vectors for software artefacts. By operating on the continuous side of the symbolic bridge, LATENT establishes a neuro-symbolic foundation for reliable and verifiable software engineering.

Part I: Scientific Proposal

1 The Scientific Vision

Programs are symbolic objects: discrete, structured, amenable to execution and formal reasoning but opaque to numerical optimization. At an AI model's input and output boundary, they inhabit a *token space*: source code text is represented as a discrete sequence of vocabulary elements. Modern AI has produced a parallel world of dense vector spaces learned by large language models, in which programs, bugs, patches, tests, and specifications implicitly exist as vectors in the latent space of models. These two worlds of symbolic tokens and numerical vectors are not yet connected as a scientific discipline.

Consider an off-by-one bug in a boundary check. Symbolic repair tools must enumerate candidate patches one at a time: replace $x < n$ by $x \leq n$, rebuild, rerun the tests, and repeat over a large discrete search space. Inside a code model, the same buggy program induces a hidden state in which $x < n$ and $x \leq n$ coexist as superposed, weighted alternatives. A gradient step can move this state toward the correct fix before a single token of the patch is written. Decoding the state yields one concrete patch, and the test suite delivers a symbolic verdict on it. This loop of continuous intervention followed by symbolic verification is what I call "software engineering in the latent space".

LATENT creates that discipline. The central vision is a duality between the symbolic space and the continuous latent space; the central bet is that the latent side is a rich yet unexploited region for engineering software. Figure 1 depicts this duality. On one side, a symbolic program maps to a latent state, and a latent state can be decoded back into a symbolic program: the two representations are two faces of the same artefact. On the other side, engineering happens as a coupled loop across the bridge: continuous intervention steers a trajectory in latent space, while a symbolic checker verifies each decoded program and returns counterexamples that trigger the next intervention. The same duality applies uniformly to bugs, patches, tests, and specifications, which is what makes it the foundational concept of the proposal.

Within this duality, the roles of the two spaces are distinct: the latent space guides, and execution decides. The latent representation is a fast, learned complement to program semantics rather than a substitute for them. It predicts execution-relevant properties of incomplete candidates [1], prioritises the next repair, test, analysis, or tool call, and signals when concrete checking is needed. Execution, tests, contracts, and program analyses remain the authoritative oracles, and each of their verdicts informs the next latent intervention.

Our thesis is that future software engineering will be performed *as a bridge between the two spaces*, as a symbiosis of neural and symbolic knowledge. Gradient-based program repair [2], latent chain-of-thought reasoning [3, 4], mechanistic interpretability of code [5–8], and representation engineering [9, 10] are early, isolated signals that this territory is real and tractable. LATENT turns these signals into a coherent, ambitious and impactful research programme.

2 Objectives

The central research question of this project is: **Can continuous latent space representations of software support reliable semantic reasoning and intervention, with a verifiable bridge to symbolic programs?**

By *continuous representations of software*, we mean real-valued vectors that encode discrete software artefacts and their properties inside an AI model. The represented objects include programs and patches, but also specifications, tests, execution behaviours, security policies, and architectural invariants. Such representations occur as hidden states in code language models. They can be varied, compared, combined, and optimised by geometric and gradient-based operations before being decoded back into symbolic artefacts. Whether this latent geometry of programs faithfully captures software semantics is the central empirical question of LATENT.

To answer it, the project pursues the following objectives:

1. **Representation: discover the latent geometry of software semantics.** Determine which properties of programs, tests, specifications, and software architectures are encoded in learned continuous representa-

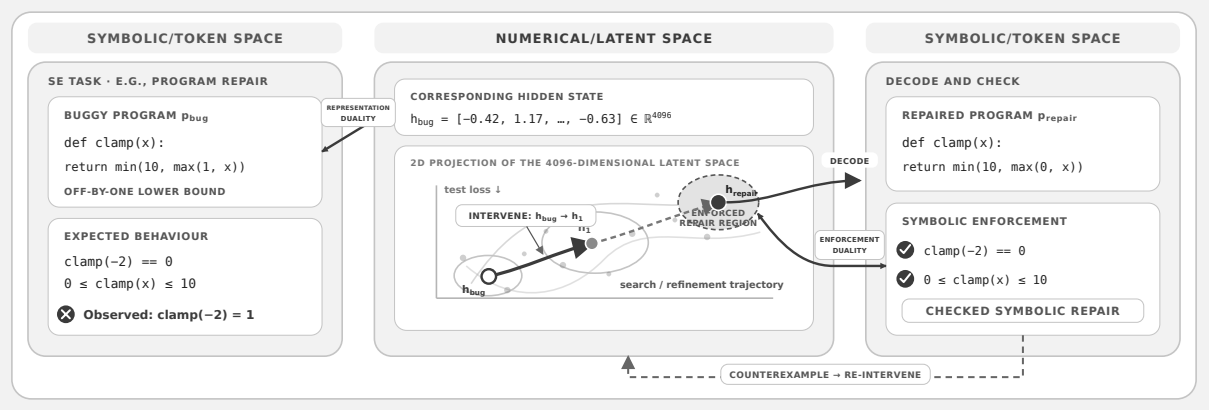

Figure 1: The LATENT vision. Program–vector duality connects a symbolic program to its latent representation. Enforcement duality couples intervention along a trajectory in latent space with symbolic checking of each new program; counterexamples trigger re-intervention. The concept generalises to patches, tests, specifications, and any software artefacts ingested by AI coding models.

tions of the latent space; where and in what form they are encoded; and whether these representations remain stable across layers, models, programming languages, and semantics-preserving transformations. In the running example: where and in what form does the model encode the boundary condition and its correctness?

2. **Intervention: make latent software semantics causally actionable.** Establish when continuous representations can be manipulated to generate, repair, or secure programs. Distinguish representations that merely correlate with a software property from those that causally control model behaviour, and develop latent space interventions that compose without destructive interference. In the running example: which latent operation reliably moves the state from $x < n$ to $x \leq n$?

3. **Verification: bridge continuous search back to symbolic guarantees.** Develop methods that map latent predictions, latent trajectories, and latent interventions to symbolic programs, tests, or counterexamples whose correctness can be checked. Characterise explicitly when latent evidence is heuristic, when it is empirically reliable, and when it supports a formal guarantee through symbolic validation. In the running example: what does the test-suite verdict on the decoded patch guarantee about the latent trajectory that produced it?

These objectives are refined into ten subobjectives (A to J, Section 3), and each subobjective is stated in Part B2 as a falsifiable claim paired with a controlled experiment. Negative results are informative because they chart where latent software semantics breaks down.

3 Research Strategy

3.1 Concepts

The project will set up the conceptual foundations of latent software engineering. We now outline the core intuitions. *Token space* is the discrete space of vocabulary sequences at a model's input and output: changing a program means changing tokens. The ingestion of a program induces latent vectors. The term *latent space* covers several related but distinct continuous objects: learned representations in a model's residual stream; continuous thought states used for multi-step reasoning; intervention directions. A *latent-space operation* is an explicit comparison or transformation of these vectors before decoding to symbolic. Examples include measuring distance, projecting onto a subspace, adding a steering direction, combining candidate states or optimising by gradients. A *latent trajectory* is the sequence of states produced as a software engineering task is performed. Its direction, speed, and crossings of latent boundaries may reveal computational and assurance

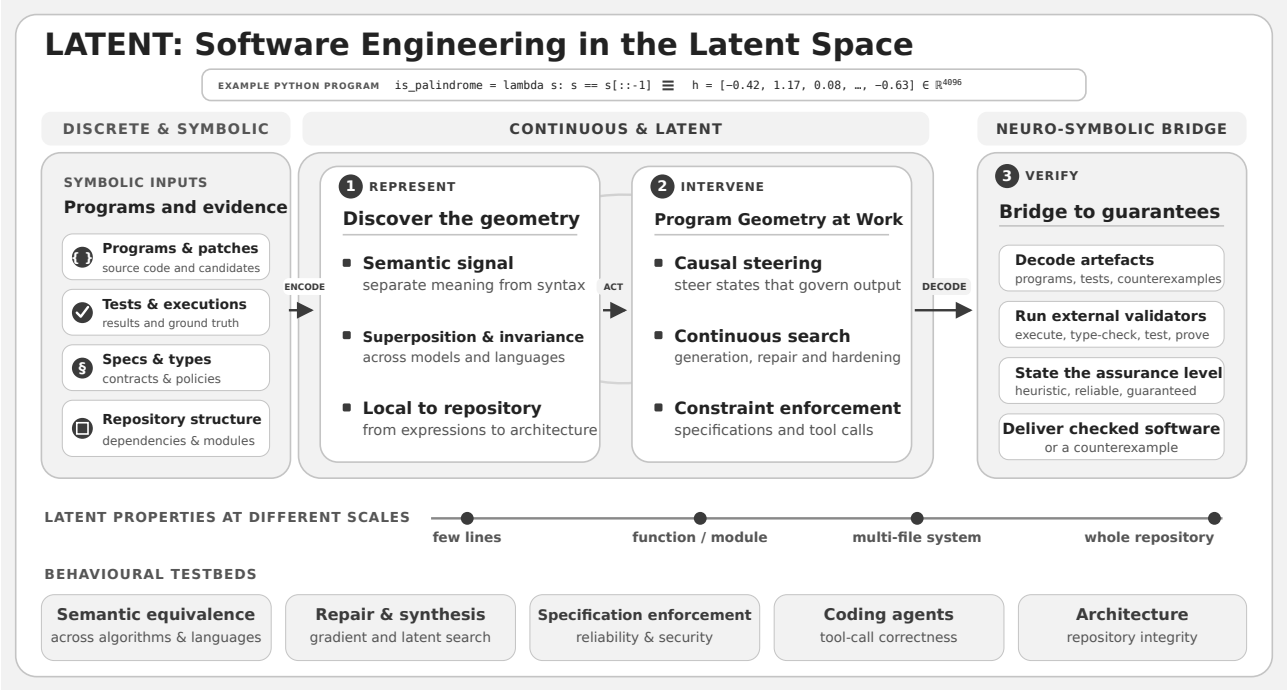

Figure 2: The LATENT research proposal. The core vision is that programs have a dual identity as symbolic programs (as we know them today) and a continuous vector representation in the latent space. The project will discover this uncharted duality and make it actionable through software engineering tasks.

properties. LATENT turns knowledge of the latent space into validated software engineering action, with program repair, semantic equivalence, specification enforcement, and architectural integrity as semantic test beds. Figure 2 connects the three objectives across scales and software-engineering testbeds.

Objective 1: Representation

The first objective asks what continuous representations in the latent space capture about a program. LATENT will test whether recovered features remain stable across layers and models. Causal experiments will then establish where those properties participate in the model's own computation.

A: Superposition of software concepts in the latent space. LATENT will study *superposition* in code-trained models. Superposition allows a model to encode more features than it has dimensions, with multiple features sharing representational capacity. LATENT investigates how this shared capacity encodes software concepts at multiple levels: programming languages and API idioms, algorithms, coding styles, error handling, correctness, and security. Early results show promise for this [5, 11]. LATENT will determine which concepts share subspaces, which interfere, and which remain recoverable under changes of language or style. Causal interventions will test whether a feature merely records a concept or controls model behaviour. The resulting map of latent representations will show where latent-space operations can be precise and when they require external checks [6, 10].

B: Semantic equivalence in the latent space. LATENT will address one of the most fundamental questions in computing: when do two programs compute the same thing? Classical semantic equivalence checking is undecidable in general and computationally intractable in practice, forcing compilers and optimisers to rely on conservative approximations. LATENT tests the hypothesis that semantically equivalent programs, regardless of algorithm, language, or surface form, occupy nearby or systematically related regions of a learned representation, as explicitly trained multilingual code spaces already demonstrate [12]. The PI has shown that a neural model can learn rewrite rules to prove equivalence between straight-line programs [13]. LATENT asks whether the geometry of the latent space itself already carries this signal. If successful, a latent equivalence measure

could prioritise candidate compiler transformations and translation pairs for subsequent symbolic validation.

C: Software architecture integrity in the latent space. LATENT will span the full scale of software semantics, from properties of a few lines of code to repository-level properties that emerge across thousands of files. At repository level, these properties include cross-file call, dependency structures, layering rules, and architectural invariants. Recent work suggests partial architectural views [14]. LATENT will locate candidate representations of software architecture, test whether they remain stable as the working context changes, and intervene on them to measure their effect on architectural decisions. This connects small-scale semantics, where a property may concern one expression, to the large-scale integrity of an entire codebase.

Objective 2: Intervention

The second objective makes identified software semantics in the latent space actionable. We will use latent representation manipulation accross various tasks incide repair, generation and agent control.

D: Superposition for program repair. Objective 1 studies superposition as a representational fact; here LATENT uses it as a computational resource. As in the running example, a hidden state retains and composes candidate patches such as $x < n$ and $x \leq n$, together with evidence about their expected behaviour, instead of committing immediately to one patch [2]. Differentiable proxies such as Gumbel-Softmax let gradients shape this superposed state before a single patch is decoded [15–17]. This establishes automated debugging in the latent space, where multiple debugging hypotheses and multiple candidate patches are explored simultaneously.

G: Latent generation from specification to implementation. LATENT asks whether the reasoning chain from a specification to an implementation can be made continuously between input and output latent vectors. There are early signals that this works: COCONUT and CODI compress planning reasoning into continuous states [3, 18], GeoMathCode identifies distinct latent subspaces for mathematical reasoning [19]. LATENT will transfer this capability to the key domain of specification-to-code generation. The latent process must decompose requirements, preserve competing implementation hypotheses, and check partial designs against invariants. It must also recognise when no correct implementation can yet be produced by analyzing latent reasoning trajectories [20].

E: Specification enforcement in latent space. The task is to generate software that satisfies requirements beyond observed input-output examples, including preconditions, security policies, type contracts, and API rules. This is an intervention problem because the latent representation of a requirement must actively constrain generation. LATENT will compile each specification into a latent constraint and steer model states toward compliant programs [9]. Residual-stream steering can convert some of this latent awareness into more secure generation without retraining [10]. LATENT will determine whether a property to enforce is represented as a direction, a region defined by proximity, a decision boundary, or a linear subspace. It will then study how such geometries can be enforced in the latent space.

F: Tool calls as latent action programs. A coding agent modifies a repository by alternating between code generation and external tools such as search, compilers, test runners, and version control. Its success depends on deciding when a tool is needed, choosing it, and supplying arguments appropriate to the repository state. Tool necessity is decodable from internal states, and activation steering can improve the decision to call a tool [21]. Specific tool identity is also linearly readable and steerable in fixed-menu, single-turn settings [22]. LATENT goes beyond correcting these local decisions. It will represent a complete tool call as a latent action: tool identity, typed arguments, preconditions, expected state change, and its place in a longer plan would all be represented as vectors. Causal interventions will control these components jointly before any call is serialised. This creates an internal semantic control plane for coding agents.

Objective 3: Verification

The third objective makes latent reasoning and computation accountable at the symbolic boundary. It treats the latent space as a predictive and control layer, while retaining symbolic execution, tests, types, contracts, and analyses as the authoritative oracles. It measures how much assurance a latent result supports.

H: Dual-space specification enforcement. Subobjective E reasons about a specification geometrically so that

it can steer code generation. Subobjective H retains the original symbolic specification as the authority. Every artefact decoded into symbolic form will therefore face the appropriate executable check: tests, types, contracts, security analysis, or architectural rules. Such mixed symbolic feedback already improves vulnerability repair [23]. The comparison between latent and symbolic enforcement will reveal where a geometric constraint survives decoding and where information is lost. When a symbolic check fails, as when the decoded boundary-check patch of the running example breaks a test, the resulting counterexample provides a precise signal for either training or for the next latent intervention. Security and reliability are thus enforced in both spaces, with continuous guidance during latent generation and symbolic judgement at the boundary.

I: Trajectory Assurance. A single latent state raises a static question: what does its location predict about the correctness or security of the program it decodes to? Latent generation and reasoning, however, follow a *trajectory* through the space, raising the question of whether the path in the latent space stays inside reliable regions or drifts through unsafe ones. This subobjective targets quality control of the entire trajectory, not the inspection of an isolated state. LATENT will map regions and trajectories in latent space to explicit assurance levels. These range from heuristic ranking to a guarantee established by an external checker. The project will test whether distance from a learned failure boundary predicts survival under decoding and execution; there is initial work showing the feasibility of statistical assurance claims for code generation [24]. It will determine which latent states should emit an executable program, a targeted test or a counterexample. The result will be an assurance-aware latent search process that jointly optimises both candidate quality and the strength of the claim that can be attached to it.

J: Durable latent states. Software knowledge must have a life beyond one prompt or engineering task. LATENT will create durable states: a persistent latent representation of an evolving codebase, think of Git for latent vectors. The durable ledger will pair each latent vector to their symbolic guarantee counterparts such as tests, contracts, security policies, and architectural rules. New validation evidence will produce a targeted intervention on the latent state, allowing reasoning to continue without reconstructing the repository's semantics from scratch. The central question is whether a continuous state can accumulate useful software knowledge without drifting away from evolving code or earlier guarantees. A positive result would establish a durable neuro-symbolic memory for software development beyond task-bounded coding agents.

3.2 Open Benchmarks, Open Evals.

Empirical science advances only as fast as its benchmarks and datasets: sound, openly shared artefacts let a community measure progress honestly and build cumulatively. LATENT is committed to producing such artefacts along two axes. The first is a suite of carefully engineered *task* benchmarks in the latent space: program repair, software architecture integrity, semantic equivalence, and specification enforcement, each with executable checks and documented protocols.

The second, and more novel, is a set of *latent-space datasets*: the vectors, states, and trajectories that programs, bugs, patches, and tests induce inside code models, paired with their symbolic outcomes. Datasets of latent vectors for software artefacts essentially do not exist today; releasing them is itself a contribution that opens the field to future researchers.

The PI has extensive experience in producing high quality research prototypes that are reused, as well as sound benchmarks and datasets that other researchers build upon.

4 Impact

Software engineering in the latent space does not exist today. LATENT will establish it as a field with concepts, methods, assurance criteria, and open benchmarks. It changes the object of software-engineering study from generated tokens to the continuous computation that precedes them. Repair, testing, specification mining, and architecture recovery gain geometric formulations tied to classical executable semantics.

For AI, software gives internal representations precise external meaning. LATENT connects model geometry to rich, checkable behaviour by distinguishing decodable information, causal control, and assurance. These results can reshape representation engineering, latent reasoning, and agent design.

The ultimate impact is a breakthrough in the quality and complexity of software that can be built with AI. Current coding agents produce local edits and rely on naive, external rejection after generation. LATENT opens a path to agents that reason over competing implementations, preserve repository-wide architecture, enforce security and reliability constraints during generation, and carry evidence for their outputs. Such agents could construct and maintain systems whose scale and semantic complexity exceed the reach of today's token-level methods. Establishing whether this vision is possible will set the research agenda for the next generation of AI-assisted software engineering.

5 Risk Management

The central risk is that software semantics may be decodable but causally unused, unstable across models, or lost when continuous states are decoded. These failures are scientifically informative. LATENT addresses them through causal intervention, cross-model and cross-language replication, adversarial controls for syntax and provenance, and executable or formal validation at the continuous–symbolic boundary.

References

[1] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. [PDF].

[2] André Silva, Gustav Thorén, and Martin Monperrus. Gradient-based program repair: Fixing bugs in continuous program spaces. *arXiv preprint arXiv:2505.17703*, 2025. [PDF] [Related].

[3] Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024. [PDF] [Related].

[4] Hanlin Zhu, Shibo Hao, Zhiting Hu, Jiantao Jiao, Stuart Russell, and Yuandong Tian. Reasoning by superposition: A theoretical perspective on chain of continuous thought. *arXiv preprint arXiv:2505.12514*, 2025. [PDF] [Related].

[5] K. Tahimic and Charibeth Cheng. Mechanistic interpretability of code correctness in LLMs via sparse autoencoders. *arXiv preprint arXiv:2510.02917*, 2025. [PDF] [Related].

[6] Zhe Yin, Xiaodong Gu, and Beijun Shen. Neuron-guided interpretation of code LLMs: Where, why, and how? *arXiv preprint arXiv:2512.19980*, 2025. [PDF] [Related].

[7] André Silva, Han Tu, and Martin Monperrus. Latent programming horizons in coding agents. Technical Report arXiv:2607.05188, arXiv, 2026. [PDF].

[8] Francisco Ribeiro, Claudio Spiess, Prem Devanbu, and Sarah Nadi. On LLMs' internal representation of code correctness. In *International Conference on Software Engineering (ICSE)*, 2026. [PDF].

[9] Lukasz Bartoszcze, Sarthak Munshi, Bryan Sukidi, J. Yen, Ze Yang, David Williams-King, Linh Le, Kosi Asuzu, and C. Maple. Representation engineering for large language models: Survey and research challenges. *arXiv preprint arXiv:2502.17601*, 2025. [PDF] [Related].

[10] Maximilian Wendlinger, Daniel Kowatsch, Konstantin Böttinger, and Philip Sperl. Security-by-design for LLM-based code generation: Leveraging internal representations for concept-driven steering mechanisms. *arXiv preprint arXiv:2603.11212*, 2026. [PDF] [Related].

[11] Yicheng He, Zheng Zhao, Zhou Kaiyu, Bryan Dai, Jie Fu, and Yonghui Yang. CodeCircuit: Toward inferring LLM-generated code correctness via attribution graphs. *arXiv preprint arXiv:2602.07080*, 2026. [PDF].

[12] Junhao Chen, Jingxuan Zhang, Jian He, Yixuan Tang, and Weiqin Zou. Bridging the programming language gap: Constructing a multilingual shared semantic space through AST unification and graph matching. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*, 2026. [PDF].

[13] Steve Kommrusch, Martin Monperrus, and Louis-Noël Pouchet. Self-supervised learning to prove equivalence between straight-line programs via rewrite rules. *IEEE Transactions on Software Engineering*, 49 (7):3771–3792, 2023. [PDF].

[14] G. Sapunov. Theory of code space: Do code agents understand software architecture? *arXiv preprint*

arXiv:2603.00601, 2026. [PDF] [Related].

[15] Matthew Macfarlane, Clément Bonnet, Herke van Hoof, and Levi Lelis. Gradient-based program synthesis with neurally interpreted languages. In *Proceedings of ICLR 2026*, 2026. [PDF] [Related].

[16] Jian Gu, Aldeida Aleti, Chunyang Chen, and Hongyu Zhang. A semantic-based optimization approach for repairing LLMs: Case study on code generation. *arXiv preprint arXiv:2503.12899*, 2025. [PDF].

[17] Zirui Li, Xuefeng Bai, Kehai Chen, Yizhi Li, Jian Yang, Chenghua Lin, and Min Zhang. Dynamics within latent chain-of-thought: An empirical study of causal structure. *arXiv preprint arXiv:2602.08783*, 2026. [PDF] [Related].

[18] Zhenyi Shen, Hanqi Yan, Linhai Zhang, Zhanghao Hu, Yali Du, and Yulan He. CODI: Compressing chain-of-thought into continuous space via self-distillation. In *Proceedings of EMNLP 2025*, 2025. [PDF] [Related].

[19] Yingji Zhang, Yong Dai, and André Freitas. GeoMathCode: Understanding interleaved math-code reasoning for geometry problem solving. *arXiv preprint arXiv:2605.25384*, 2026. [PDF].

[20] Hanwen Du, Yuxin Dong, and Xia Ning. Latent thinking optimization: Your latent reasoning language model secretly encodes reward signals in its latent thoughts. In *Proceedings of the International Conference on Learning Representations*, 2026. [PDF].

[21] Youjin Wang, Run Zhou, Rong Fu, Shuai Cao, Hongwei Zeng, Jiaxuan Lu, Sicheng Fan, Jiaqi Zhao, and Liangming Pan. ASA: Backbone-training-free representation engineering for tool-calling agents. *arXiv preprint arXiv:2602.04935*, 2026. [PDF] [Related].

[22] Zekun Wu, Ze Wang, Seonglae Cho, Yufei Yang, Adriano Koshiyama, Sahan Bulathwela, and Maria Perez-Ortiz. Tool calling is linearly readable and steerable in language models. *arXiv preprint arXiv:2605.07990*, 2026. [PDF].

[23] Yifan Zhang, Jieyu Li, Kexin Pei, Yu Huang, and Kevin Leach. SynthFix: Adaptive neuro-symbolic code vulnerability repair. *arXiv preprint arXiv:2604.17184*, 2026. [PDF].

[24] Senrong Xu, Yuhao Tan, Yanke Zhou, Guangyuan Wu, Zenan Li, Yuan Yao, Taolue Chen, Feng Xu, and Xiaoxing Ma. Uncertainty quantification for LLM-based code generation. *arXiv preprint arXiv:2605.12201*, 2026. [PDF].

Martin Monperrus

Full Professor of Software Technology, KTH Royal Institute of Technology, Stockholm, Sweden

Education

2008 **PhD** in Computer Science, *with highest distinction*, Université de Rennes 1, France.
Thesis: *Model-driven Generative Development of Measurement Software*.
Supervisor: Jean-Marc Jézéquel.

2004 **MSc** in Computer Science, *ranked 2nd, with highest distinction*, Université de Technologie de Compiègne, France.
Thesis: *Manifolds: Beyond Locality*.
Supervisor: Yoshua Bengio (Turing Award 2018; pioneer of deep learning).

Current and Previous Positions

2017–now **Full Professor of Software Technology**, KTH Royal Institute of Technology, Stockholm, Sweden. Holder of a Wallenberg Foundation WASP Chair (2017; one of 10 such chairs in Sweden). Leads the Software Technology research group (~10 members).

2016–2018 **Co-founder & Chief Scientific Officer**, Makitoo (startup), Sweden. Raised 450 k€ in private capital.

2011–2017 **Associate Professor**, Université de Lille / Inria, France.

2008–2011 **Research Associate**, TU Darmstadt, Germany.

2005–2008 **Research Assistant**, ENSIETA, Brest, France.

Key Qualifications

LATENT bridges two disciplines, and the PI is one of the very few researchers worldwide with a deep track record on both sides of the bridge. On the continuous side, the PI was trained in representation learning by Yoshua Bengio (MSc thesis on the geometry of learned manifolds, 2004). On the symbolic side, the PI has spent seventeen years of research leadership making learning work for software engineering: statistical code completion (2009), structural code transformation (2014), neural program repair (2018), gradient-based repair in continuous program spaces (2025), and probing of coding-agent latent states (2026). This combination of representation-learning roots, symbolic software-engineering leadership, and open, reusable research infrastructure is precisely the profile LATENT requires.

Selected Peer Recognition

- **IEEE Fellow 2026** — "for pioneering machine learning use in code assistance and program repair, impacting millions of developers"; at most 0.1% of IEEE voting members are elevated annually.
- **ACM SIGSOFT Impact Paper Award 2024** — for pioneering machine learning for code assistance and paving the way for today's intelligent code-assistance tools.
- **IEEE TSE Best Paper Award 2024** — for VRepair, selected from the 2023 volume of the field's flagship journal.
- **ASE Most Influential Paper Award 2024** — for GumTree's impact over the decade following publication.
- **KTH PhD Supervisor of the Year 2023/24** — selected through a doctoral-student nomination and jury process across KTH.

Invited keynotes.

- **2026:** Big Data 2026 (scheduled) and Ada-Europe 2026, "Bootstrapping Coding Agents: The Specification Is the Program."
- **2026:** Rekt Security Summit, "Software Supply Chain Security of Web3."
- **2025:** APSEC, "Software Supply Chain Security of Web3"; STEW, "Recent Progress in Neural Program Repair."
- **2022:** AIST/ICST, "Sequence-to-Sequence Learning for Software Testing."

Research Outputs (up to 10)

Google Scholar profile ORCID: 0000-0003-3505-3383. Citation counts below are from Google Scholar, July 2026.

[1] A. Silva, H. Tu, **M. Monperrus**. "Latent Programming Horizons in Coding Agents." Technical report, arXiv:2607.05188, 2026. [link]
Key preliminary result. *Demonstrates that coding-agent hidden states linearly encode parse validity, test adequacy, and bug presence, and predict outcomes up to 25 steps ahead. PI initiated and supervised the project. Directly validates LATENT's central hypothesis that software properties are decodable from LLM residual streams.*

[2] A. Silva, G. Thorén, **M. Monperrus**. "Gradient-Based Program Repair: Fixing Bugs in Continuous Program Spaces." Technical report, arXiv:2505.17703, 2025. [link]
Key preliminary result. *Formulates repair as continuous optimisation in a differentiable program space. PI initiated and supervised the project. Direct feasibility evidence that gradient-based navigation of a program space is computationally tractable.*

[3] Z. Chen, S. Fang, **M. Monperrus**. "Supersonic: Learning to Generate Source-Code Optimizations in C/C++." *IEEE Transactions on Software Engineering*, 2024. [link]
Generates source-level performance optimisations with a neural code model. PI supervised and co-designed the study. Relevant: demonstrates that non-functional program properties can be improved with learned code transformation.

[4] S. Kommrusch, **M. Monperrus**, L.-N. Pouchet. "Self-Supervised Learning to Prove Equivalence Between Straight-Line Programs via Rewrite Rules." *IEEE Transactions on Software Engineering*, 2023. [link]
Self-supervised learning of rewrite rules for proving program equivalence. PI supervised and co-designed the approach. Directly supports LATENT's semantic-equivalence and symbolic-verification work.

[5] Z. Chen, S. Kommrusch, **M. Monperrus**. "Neural Transfer Learning for Repairing Security Vulnerabilities in C Code." *IEEE Transactions on Software Engineering*, 2023. **274 citations.** [link]
Introduced VRepair to repair security vulnerabilities. PI supervised and co-designed the methodology. Evidence for LATENT's security and specification-enforcement objectives. [**IEEE TSE Best Paper Award 2024**]

[6] H. Ye, M. Martinez, **M. Monperrus**. "Neural Program Repair with Execution-based Backpropagation." *Proc. ICSE*, 2022. **274 citations.** [link]
Integrates compilation and test-execution signals into neural repair. PI initiated the project and supervised the design, seed for later RL with verifiable rewards. Direct precedent for LATENT's coupling of continuous intervention with executable, symbolic feedback.

[7] M. Bruch, **M. Monperrus**, M. Mezini. "Learning from Examples to Improve Code Completion Systems." *Proc. ESEC/FSE*, 2009. **537 citations.** [link]
Demonstrated corpus-trained statistical code completion before neural code models even exist. PI co-designed the learning approach and led the scientific content. Foundational evidence that source code contains learnable regularities, the core premise of LATENT. [**ACM SIGSOFT Impact Paper Award 2024**]

[8] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, **M. Monperrus**. "SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair." *IEEE Transactions on Software Engineering*, 2019. **745 citations.** [link]
Established end-to-end sequence-to-sequence repair on real bug-fix data. PI initiated and supervised the project. LATENT generalises this discrete neural-repair paradigm to continuous representations and multiple SE tasks.

[9] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, **M. Monperrus**. "Fine-grained and Accurate Source Code Differencing." *Proc. ASE*, 2014. **878 citations.** [link]
Introduced GumTree's scalable AST-mapping approach for fine-grained source-code differencing. PI co-developed the work and supervised the evaluation. Precise symbolic change representations support LATENT's analysis of program transformations. [**ASE Most Influential Paper Award 2024**]

[10] **M. Monperrus**. "Automatic Software Repair: A Bibliography." *ACM Computing Surveys*, 2017. **703 citations.** [link]

Sole-authored synthesis that gave program repair a common taxonomy and articulated its benchmarks and open problems. Establishes the PI's command of the symbolic repair landscape that LATENT connects to continuous representations.

Ability to Conduct Ground-Breaking Research

The PI has repeatedly turned unconventional research hypotheses into results that changed software-engineering research and practice. The PI's research outputs have 14 000+ citations and an H-index of 57 (July 2026), but the strongest evidence is the independent recognition of several different contributions.

AI on code with lasting practical impact. The 2009 FSE paper (output [7]; 537 citations) demonstrated that code corpora could improve completion when mainstream tools only relied on type information and hand-designed ranking rules. Fifteen years later, the ACM SIGSOFT Impact Paper Award recognised it for paving the way for today's intelligent code-assistance tools. What was a fringe hypothesis in 2009 is now a multi-billion-dollar industry: learned code assistants such as GitHub Copilot and Cursor are used by tens of millions of developers. IEEE elected the PI Fellow "for pioneering machine learning use in code assistance and program repair, impacting millions of developers."

Research infrastructure that changed how source code is studied. GumTree (output [9]; 878 citations) made fine-grained AST mappings practical at scale. It is now maintained as open infrastructure, integrated into developer and research tools, and used as a methodological foundation in hundreds of studies. The ASE Most Influential Paper Award 2024 recognised its impact over the decade after publication. The Spoon library (447 citations, 1 000+ GitHub stars) has a similar impact for source-code analysis and transformation; it is reused in 100+ open-source projects and industrial tools, and has become the top infrastructure for Java program-analysis research. Together they are evidence of the PI's ability to create a durable technical substrate on which a community builds.

A sustained programme in learned program repair. SequenceR (output [8]; 745 citations) first established end-to-end neural repair on real bug-fix data. The programme then progressed from prediction to execution-grounded learning and security: execution-based backpropagation (output [6]) incorporated compilation and testing into learning, while VRepair (output [5]) transferred knowledge from ordinary bug fixes to vulnerabilities. VRepair received the IEEE TSE Best Paper Award 2024. Automatic program repair, which the PI's own survey (output [10]) helped systematise, has since grown from a handful of prototypes into an established subfield with hundreds of papers per year and industrial deployments at Meta, Google and Bloomberg. Together, these results show the ability to originate a research direction, deepen it methodologically, extend it to high-stake domain, and contribute to changing the world.

The next frontier is already producing evidence. Open-weight code models expose residual streams at research scale for the first time, and interpretability methods have matured on natural language while software semantics remains uncharted. The PI's group has produced the first evidence about that: outputs [1] and [2] show respectively that software properties are decodable from coding-agent residual streams and that repair can be formulated as continuous optimisation. LATENT starts from phenomena demonstrated in the PI's own group and asks the substantially more ambitious questions of causality, generality, and verifiable intervention.

Creative and Original Thinking

The PI's contributions show one repeated creative move — formulating a novel fundamental SE problem — pursued each time years before the community believed it viable.

2004 — the formative instinct. The PI's MSc thesis *Manifolds: Beyond Locality*, supervised by Yoshua Bengio (Turing Award 2018), studied the structure of learned continuous manifolds. The conviction that the right continuous representation is the intellectual seed of LATENT.

2009 — from types to learned code regularities. While the field ranked completions with hand-written type rules, the FSE paper instead trained models on open-source corpora to predict code — a decade before neural code models existed, and now the premise of a multi-billion-dollar industry.

2014 — from lines to structural changes. Spoon and GumTree re-expressed programs and their edits as operations over abstract-syntax trees, replacing line-based diffing with structural reasoning about what changed — the representation on which code analysis research now builds.

2018 — from symbolic search to learned repair. SequenceR was the first ever to learn repair from real bug-fix commits; the PI's subsequent work incorporated execution feedback and transferred knowledge from ordinary fixes to security vulnerabilities, originating neural program repair as a research direction from an unconventional initial hypothesis.

2025–2026 — from tokens to continuous program spaces. GBPR made repair amenable to gradient-based search over a differentiable program representation. Latent Programming Horizons then showed that coding-agent residual streams encode present software properties and anticipate future outcomes. These complementary preliminary results motivate LATENT's central step: treating continuous representations as a new substrate for software engineering.

Scientific Expertise and Capacity to Successfully Execute the Project

LATENT requires an unusual combination of neural code modelling, program analysis, execution-based evaluation, and scientific leadership. The PI has built this combination over 17 years and has already assembled the people and infrastructure needed to start the project.

Representation Output [1] probes software properties in coding-agent residual streams; outputs [3] and [7] establish learned representations of functional and non-functional code regularities.

Intervention Outputs [2], [5], [6], and [8] demonstrate gradient-based, sequence-to-sequence, transfer-learning, and execution-guided program repair.

Verification Output [4] addresses semantic equivalence; outputs [6] and [9], together with Spoon, connect learned transformations to executable tests and structured symbolic changes.

Execution Seven current PhD students, including two already working on latent-space SE, have access to WASP/NAISS GPU infrastructure and established program-analysis platforms.

- **Leadership at ERC scale:** 21 competitive grants totalling 12.5 M€, including CHAINS (SSF, main PI, 31.0 MSEK, 2022–27), WASP AI Chair (Wallenberg, 2.5 M€, 2017–27), Digital Futures Flagship (0.54 M€, 2023–32), WASP PhD grants incl. A. Silva, the student who produced the preliminary results (outputs [1] and [2]).
- **Research-team development:** 9 graduated PhDs, 5 co-supervised, 7 current; 8 former students now hold faculty positions: Durieux (TU Delft), Ye (UCL), Madeiral (UFPE), Martinez (UPC Barcelona), Xuan (Wuhan), Yu (Shandong), Bartel (Umeå), Harrand (Stockholm).

Community Contributions

Editorial leadership. Associate Editor, IEEE Transactions on Software Engineering (2024–present), the field's flagship journal; Associate Editor, Springer Empirical Software Engineering (2015–2023).

Open-source infrastructure. Spoon: Java source-code analysis and transformation library; co-developed and actively maintained since 2012; 1 000+ GitHub stars; used in 50+ open-source projects and industrial tools at Google, HSBC and others. GumTree: widely reused fine-grained source-code differencing infrastructure, integrated in IDEs and used in hundreds of research papers.

Open science commitment. Co-founded the Open Science Initiative at *Empirical Software Engineering* (2019), establishing a journal-wide policy encouraging artifact availability (code, data, benchmarks) for all accepted papers. The PI's group systematically releases research artefacts through GitHub or Zenodo.

Research–practice bridge. Co-organises the annual CHAINS Workshop on software supply-chain security (2023–present), bringing together industry, academia, and government security stakeholders. Holder of Ethereum Foundation Academic Grant (2024) for research on critical blockchain infrastructure.