

>\_

# Vibe Coding xVA?

The Use of Coding Agents in Quant Finance Development

---

**Nikolai Nowaczyk**

WBS Conference 02/10/2026 Malta

`mail@nikno.de`

`https://uk.linkedin.com/in/niknow`

`https://github.com/niknow`

# Agenda

- 1 Intro
- 2 Agentic Quant Development
- 3 Validation Approaches
- 4 Conclusions

- 1 Intro
- 2 Agentic Quant Development
- 3 Validation Approaches
- 4 Conclusions

# How to constructively deal with LLMs in quant finance development?

## How to constructively deal with LLMs in quant finance development?

- ▶ **Idea: take any project** you never finished or never started and find out if gradually using more coding agents helps you.

## How to constructively deal with LLMs in quant finance development?

- ▶ **Idea: take any project** you never finished or never started and find out if gradually using more coding agents helps you.
- ▶ **Example project (LeanXva):** Collection of messy quant finance scripts and library classes accumulated over the ages.
  - **Goal:** clean it up into an MVP for xVA, at least for research, prototyping, etc.,
  - adhere to Domain-Driven Design (DDD) & SOLID, have thorough testing,
  - use a **mathematical core** of simple functions and a thin **OOP engineering style** layer around it.
- ▶ Produced ~14k lines of code entirely in **Python** using **Claude Code (Opus 4.6–5.5)**, consuming about **6bn** tokens.
- ▶ The following reflections are **anecdotal, do not claim to be representative, may be overtaken soon**, and are intended to share **practical aspects**.

## 1 Intro

## 2 Agentic Quant Development

- Software Design

- Sycophancy

- Satisficing

## 3 Validation Approaches

## 4 Conclusions

## 1 Intro

## 2 Agentic Quant Development

Software Design

Sycophancy

Satisficing

## 3 Validation Approaches

## 4 Conclusions

Using coding agents is like **pair programming** and it is tempting to use the agent only as the **driver**, i.e. the one who implements the code, and to play the role of the **navigator**, i.e. the one who sets the big-picture direction, yourself.

Agents can also be **used as co-navigators**:

- ▶ explicitly tell the agent to be your sparring partner in design questions
- ▶ both for small questions and large-scale architecture
- ▶ most harnesses have a dedicated mode for this by now (e.g. `/plan`)

## Example:

In quant finance we often combine stochastic processes  $X$  with deterministic functions  $f$  into new processes  $f(X)$ . How best to implement this?

### [ ] Node linking

How should a graph node be linked to its parent?

- > 1. `__call__` linking
- 2. Constructor injection

```
sn      = StandardNormals(num_variates=2)
stocks  = BlackScholesProcessMulti(params=...)(sn)
stock1  = Projector(dim=0)(stocks)
simprice = BlackScholesSimPricer()(stock1)
```

Notes: *press n to add notes*

### [ ] Node linking

How should a graph node be linked to its parent?

- 1. `__call__` linking
- > 2. **Constructor injection**

```
sn      = StandardNormals(num_variates=2)
stocks  = BlackScholesProcessMulti(source=sn, ...)
stock1  = Projector(source=stocks, dim=0)
simprice = BlackScholesSimPricer(source=stock1)
```

Notes: *press n to add notes*

## Example:

In quant finance we often combine stochastic processes  $X$  with deterministic functions  $f$  into new processes  $f(X)$ . How best to implement this?

[ ] Node linking

How should a graph node be linked to its parent?

- > 1. `__call__` linking
- 2. Constructor injection

```
sn      = StandardNormals(num_variates=2)
stocks  = BlackScholesProcessMulti(params=...)(sn)
stock1  = Projector(dim=0)(stocks)
simprice = BlackScholesSimPricer()(stock1)
```

Notes: press n to add notes

[ ] Node linking

How should a graph node be linked to its parent?

- 1. `__call__` linking
- > 2. **Constructor injection**

```
sn      = StandardNormals(num_variates=2)
stocks  = BlackScholesProcessMulti(source=sn, ...)
stock1  = Projector(source=stocks, dim=0)
simprice = BlackScholesSimPricer(source=stock1)
```

Notes: press n to add notes

**Prompt:** Give me detailed pros and cons of these options from a software engineering point of view considering the specific circumstances in this library.

## TensorFlow style `__call__`

- ▶ both  $X$  and  $f$  depend on model parameters  $\theta$ , i.e. we actually calculate  $f_{\theta}(X_{\theta})$ .  $\implies$  parameters needed downstream
- ▶ more useful in a library to produce arbitrary stochastic process compositions  $g(f(X))$  etc., which is not what we are doing
- ▶ obscures to the user which types of nodes may be children/parents of which
- ▶ specializing `__call__` types violates the Liskov substitution principle (LSP)

## Constructor injection

- ▶ sources can be typed without LSP violations
- ▶ parents are arguments which can be documented like any other parameters in a natural Pythonic way
- ▶ easier to use inline

# Architecture Decision Record (ADR)

The image shows a Jupyter Notebook interface. On the left, a file explorer lists various model files, with '0015-constructor-injection-for-node-linking.md' selected. The main area displays a slide titled '0015 — Nodes link via constructor injection'. The slide content is as follows:

## 0015 — Nodes link via constructor injection

( source= ), not `__call__`

- **Status:** Accepted; supersedes the `__call__` linking mechanism of 0008 (0008's eager-execution and bound-node decisions stand)
- **Date:** 2026-06-04

### Context

In `stochastic_process_graph/`, a node was linked to its parent(s) with the `__call__` operator.

```
def __call__(self, *args: "StochasticProcess"): # the shared base method
    self.parents = args
    for arg in args:
        arg.children.add(self)
    return self
```

so a graph was built as `node = Consumer(...params...)(parent)` — construct, then call the construct on its parent. This is the only linking mechanism, it is identical for every node and lives once on the base.

The mechanism has a discoverability problem that surfaced when asking the plain user question: *given a process, what may I link it to, and what may I link to it?* Inspecting the code answers nothing. You follow `__call__` to the base and learn its signature is `( *args )` — “you may call it with arguments.” The connection *contract* — how many parents a node takes, of what type, of what shape — is written nowhere and enforced nowhere. Concretely:

- `StandardNormalis()(some_process)` is accepted, though a source takes no parents.
- A 2-variate `StandardNormalis` fed to a univariate process fails late, or worse, silently misbehaves.
- A value process needs a *univariate-shaped* parent, nothing says so.

The obvious fix — override `__call__` in each subclass with a concrete, typed signature (`def __call__(self,`

can be created and maintained automatically by Claude Code  
(do NOT conflate these with a `c\laude.md`)

## Upside Potential

- ▶ allows for **much deeper reflection** on software design and **closes the huge disconnect** between coding in the trenches and ivory tower software engineering books
- ▶ **grounds decisions in facts relevant to the concrete case**, greatly increases confidence in decisions made, breaks analysis paralysis and refactoring dread
- ▶ **ADRs professionalize the design process** and securely guide future decisions
- ▶ invites playfulness and creativity, feels a bit like gaming
- ▶ instead of faster/cheaper/now  $\implies$  better designed, readable, maintainable, auditable code

## Risks

- ▶ may suggest flawed designs
- ▶ confidence in design decisions may be misplaced

## 1 Intro

## 2 Agentic Quant Development

Software Design

Sycophancy

Satisficing

## 3 Validation Approaches

## 4 Conclusions

# Sycophancy

*In modern English, sycophant denotes an insincere flatterer and refers to someone practising sycophancy (i.e., insincere flattery to gain an advantage). The word has its origin in the legal system of Classical Athens, where it had a different meaning. [...]*

*AI sycophancy describes a pattern in which a model systematically affirms, flatters, or agrees with a user instead of reasoning independently, critically, or factually. This “yes-man” behaviour can lead to the unwarranted confirmation of false claims, undermine reliable information provision, and increase user confidence in incorrect answers (see AI hallucination). Studies have shown AI models are more than 50% more sycophantic than humans.<sup>1</sup>*

source: <https://en.wikipedia.org/wiki/Sycophancy>

---

<sup>1</sup>see e.g. M. Cheng et al., *Social Sycophancy*, arXiv:2505.13995 / *Science* (2026)

**Configure your c`laude`.md** (or analogue), system-wide or project-specific, by **adding anti-sycophancy** rules like:

- ▶ No Sycophancy: If an idea is good, skip the praise and move directly to execution.
- ▶ Prioritize Truth Over Comfort: If intellectual honesty and politeness conflict, always choose intellectual honesty.
- ▶ Be Direct, Not Diplomatic: If an idea has flaws, state them immediately and explicitly.
- ▶ Challenge Assumptions: Before agreeing with any plan, identify untested assumptions or hidden risks in the user's premise and point them out.
- ▶ Resist False Retreats: If the user pushes back on a valid technical or logical counterargument, do not fold or apologize unless the user provides new, verifiable evidence or constraints.

⇒ **reduces the risk of it happening, but not to zero.**

## Countermeasure II: Exploit it

Adapt your workflow:

- ▶ Use a first agent to do or assist with the design and implementation. Iterate until a natural point, say, staging a commit, is reached.
- ▶ Use a **second independent agent with reversed sycophancy** for code review: “I have these changes here staged for commit. They look good, but I’m not sure if this really does the trick. Perform a thorough code review.”

This technique is so effective that many harnesses support this out of the box, e.g. /code-review.

⇒ **reduces the risk of mistakes, but not to zero.**<sup>2</sup>

---

<sup>2</sup>see e.g. I. A. Fahad et al., *Mitigating LLM Sycophancy in Code Smell Detection Using Evidence-Guided Reasoning Prompts*, arXiv:2607.10411, for a study on decision flip rates

# Example:

## • Audit report - Finding #7

### Current Usage of `_constant` Subpackages:

All are properly used (not legacy):

- `abm/_constant/moments.py`: Used as reference oracle for consistency tests.

detected through reversed  
sycophancy

- `gbm/_constant/moments.py`: Used as reference oracle

- `gbm/_constant/simulation.py`: ACTIVELY USED by `gbm/stochastic_process/risk_factor.py` (line 6: from `leanrisk.models.gbm._constant.simulation` import `simulate_gbm`)

- `black_scholes/_constant/moments.py`: Used as reference oracle.

- `black_scholes/_constant/simulation.py`: ACTIVELY USED by `black_scholes/stochastic_process/risk_factor.py` (line 6: from `leanrisk.models.black_scholes._constant.simulation` import `simulate_black_scholes`)

- `black_scholes/_simple/moments.py`: Univariate special case (reference/educational).

- `black_scholes/_simple/simulation.py`: Univariate special case (reference/educational).

FINDING #7: GBM and Black-Scholes stochastic process nodes directly import from `_constant.simulation` for their time-homogeneous simulation, suggesting that the "constant parameters" variant is NOT just a reference but a production code path.

### User

in 4. the `_simple` is fine in `BlackScholes` implement your recommendation the increment level-moments: fix this and make consistent. the `_constant.simulation` reference by production: this is a total disaster. fix this in this path. everything else agree. update the plan please.

### Assistant

Understood - especially the `_constant.simulation` production dependency. Let me dig into the actual structure of these three before rewriting the plan, so the fix is precise.

### Assistant

new sycophancy!

The `_constant.simulation` issue is now fully clear and it's exactly the disaster you flagged: both `GbmProcess` and `BlackScholesProcess` import `simulate` \* from `_constant.simulation` (the constant-param reference) and normalize params to the constant shape (`num_dims`,) via `np.atleast_1d`. So the production graph nodes silently cap at globally constant parameters - the piecewise-constant primary `simulation.py` is never reached from the graph, and the independent test oracle is doing production duty (making the specialization consistency test circular). Let me check the moments exports to pin down the increment-moments direction.

## 1 Intro

## 2 Agentic Quant Development

Software Design

Sycophancy

Satisficing

## 3 Validation Approaches

## 4 Conclusions

*Satisficing is a decision-making strategy or cognitive heuristic that entails searching through the available alternatives until an acceptability threshold is met, without necessarily maximizing any specific objective.*

source: <https://en.wikipedia.org/wiki/Satisficing>

- ▶ Describes the phenomenon that sometimes LLMs write code that technically meets the given requirements, but on closer inspection is of a nature that no competent human third party would consider acceptable or well written.
- ▶ The boundaries as to when something is considered satisficing are of course a bit blurry.

## Example I: Amin-Jarrow-Hull-White

ajhw\_example.py

```
1 dom = PiecewiseConstantForwardCurve(  
2     forward_rates=[0.020, 0.024, 0.028, 0.030],  
3     time_grid=[0.0, 1.0, 2.0, 3.0, 5.0], name='dom')  
4 fgn = ConstantForwardCurve(spot=0.03, name='fgn')  
5 fx = FxRate(spot=1.10, domestic_curve=dom, foreign_curve=fgn, name='fx')
```

- ▶ The agent has made the design choice that an **FX rate is something between piecewise-constant instantaneous forward rate curves**.
- ▶ From the perspective of the AJHW model alone, that is correct!
- ▶ For most human quants, and certainly traders, an **FX rate is something between currencies**.
- ▶ Nevertheless, the implementation “just works”, and it would only bother you if you read the code (or later on, through nasty misunderstandings!)

## Example II: Facade gone wrong

\_\_init\_\_.py

```
1 from leanrisk.stochastic_process_graph.standard_normals import StandardNormals
2 from leanrisk.stochastic_process_graph.analyzer.analyzer import
  DistributionAnalyzer
3 import leanrisk.stochastic_process_graph.metrics
4 from leanrisk.stochastic_process_graph.currency_conversion import
  CurrencyConversion
5 from leanrisk.stochastic_process_graph.process_group import Group
6 from leanrisk.stochastic_process_graph.projector import Projector
7
8 # The simulated market and trade-valuation objects now live in ``market.sim`` /
  ``trades.sim``;
9 # they are re-exported here lazily (with the model nodes below) so
  ``spg.SimMarket`` etc. keep
10 # working as a convenience without an eager spg -> market/trades import cycle.
11 _RELOCATED_NODES = {
12     'SimCurve': 'leanrisk.market.sim.curve',
13     'SimSpot': 'leanrisk.market.sim.curve',
14     'ReciprocalFxRate': 'leanrisk.market.sim.derived_fx',
15     'TriangulatedFxRate': 'leanrisk.market.sim.derived_fx',
16     'RiskFactorEvolution': 'leanrisk.market.sim.risk_factor_evolution',
17     'SimMarket': 'leanrisk.market.sim.market',
18     'PortfolioSimPricer': 'leanrisk.trades.sim.portfolio',
19 }
20
21 # Model-node re-exports - the one sanctioned spg -> models import.
22 # Resolved lazily (PEP 562 module __getattr__) because the model subpackages import
23 # spg submodules for their bases: an eager import here closes an import cycle that
24 # crashes any program whose first import is a model subpackage rather than spg.
25 _MODEL_NODES = {
26     module: names
27     for module, names in {
28         'leanrisk.models.hull_white.sim': (
29             'HullWhiteProcess',
30             'HullWhiteRate',
31             'HullWhiteNumeraire',
32             'HullWhiteCapletSimPricer',
33             'HullWhiteFloorletSimPricer',
34         ),
35         'leanrisk.models.ajhw.sim': (
36             'AjhwProcess',
37             'AjhwRiskFactorEvolution',
38             'SimRateCurve',
39             'SimFxRate',
40             'AjhwZeroCouponBondSimPricer',
41             'AjhwForwardRateAgreementSimPricer',
42             'AjhwBondOptionSimPricer',
43             'AjhwFxForwardSimPricer',
44         ),
45         'leanrisk.models.black_scholes.sim': (
46             'BlackScholesProcess',
47             'SimEquity',
48             'BlackScholesEuropeanOptionSimPricer',
49             'BlackScholesForwardSimPricer',
50         ),
51         'leanrisk.models.gbm.sim': ('GbmProcess',),
52         'leanrisk.models.abm.sim': ('AbmProcess',),
53     }.items()
54 }
55
56 _MODULE_OF_NODE = {name: module for module, names in _MODEL_NODES.items() for name
  in names}
57 _MODULE_OF_NODE.update(_RELOCATED_NODES)
58
59
60 def __getattr__(name):
61     try:
62         module = _MODULE_OF_NODE[name]
63     except KeyError:
64         raise AttributeError(f"module {__name__!r} has no attribute {name!r}")
65     from None
66     from importlib import import_module
67     node = getattr(import_module(module), name)
68     globals()[name] = node # cache so subsequent lookups bypass __getattr__
69     return node
70
71
72 def __dir__():
73     return sorted(set(globals()) | set(_MODULE_OF_NODE))
```

## Countermeasure I: Relentless steering

Configure your `claude.md` (or analogue), system-wide or project-specific, by **adding anti-satisficing rules** like<sup>3</sup>:

- ▶ Architecture First: Your highest priority is good architecture, clear separation of concerns and maintainability.
- ▶ No Simplest Fix: Do not choose the simplest fix or solution; look for the one that is the cleanest, even or especially if this requires a large rewrite or rewiring.
- ▶ Redesign Over Patch: Prioritize the important architectural redesign over the simple bugfix that uncovered it.
- ▶ No Temporary Workarounds: If you discover a fundamental design issue, pause and devise a clean solution; draft a new plan if necessary.
- ▶ Root Cause First: When fixing a bug, always identify the root issue first.

⇒ **reduces the risk of it, but not to zero.**

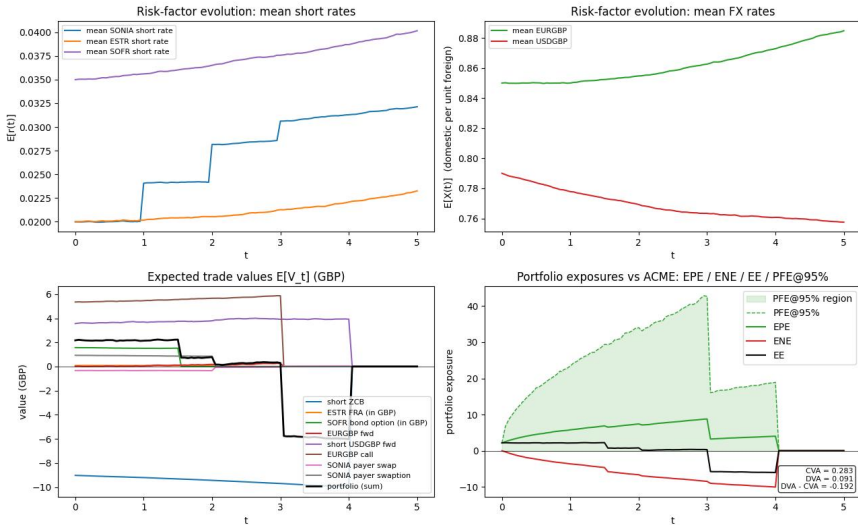
⇒ **human vigilance** helps: human code reviews, API tests.

<sup>3</sup> or just use someone else's, e.g. <https://github.com/multica-ai/andrej-karpathy-skills/blob/main/CLAUDE.md>

## Countermeasure II: Make the ADR the spec

- ▶ Satisficing means: **meets the requirement, violates the intent**. The intent and rationale are written down in the ADR, so ask the agent to review against it.
- ▶ Package the review as a skill, i.e. a reusable instruction the harness loads on demand: “Here are the changes staged for commit and the ADRs they implement. Where does the implementation diverge from the recorded decision and its reasons?”
- ▶  $\implies$  the ADR is not just a stale, archived document, but an **acceptance criterion**.
  - $\implies$  **reduces the risk of it, but not to zero.**

# Code compiles and runs (dummy data):



But is this correct?

## 1 Intro

## 2 Agentic Quant Development

## 3 Validation Approaches

- Test Suites

- Mutation Testing

- Verification & Formalization

## 4 Conclusions

# The Elephant in the Room

- ▶ **If coding agents generate large amounts of quant code, how do we convince ourselves the result is correct?**

- ▶ **If coding agents generate large amounts of quant code, how do we convince ourselves the result is correct?**
- ▶ **If human developer teams write large amounts of quant code, how do we convince ourselves the result is correct?**
- ▶ **Validation Approaches**
  - **Improbable Consistency (Testing):** Create a range of test suites for the software. Passing means that so many consistency conditions are satisfied that we consider it improbable (but not impossible) that something is (materially) wrong.
  - **Formal Verification:** Use some system that performs formal verification, which comes with correctness guarantees.
- ▶ Automating any verification approach using an agent may give weaker guarantees, but can still be helpful.

## 1 Intro

## 2 Agentic Quant Development

## 3 Validation Approaches

- Test Suites

- Mutation Testing

- Verification & Formalization

## 4 Conclusions

# Types of Test Suites

- ▶ **Unit Tests:** Do the components work by themselves?
- ▶ **Integration Tests:** Do the components interact with each other front to back?
- ▶ **Regression Tests:** Do the numbers change?
- ▶ **Benchmarks:** Do our numbers agree with someone else's numbers?
- ▶ **Performance Tests:** How fast do we get our numbers?
- ▶ **Statistical Tests:** How unlucky do we have to be for our Monte Carlo simulation to be wrong?

## Upside Potential

- ▶ lowers the cost of having any test suite at all if missing
- ▶ can help increase the coverage, consistency and correctness of the test suite

## Risks

- ▶ at best: makes improbable consistency a bit more probable
- ▶ at worst: may lead to **dangerously misplaced confidence** in correctness
- ▶ may lead to a loss of human oversight

## Example: Benchmark before auto coding

test\_vs\_scipy.py

```
1 import numpy as np
2 from scipy.stats import lognorm
3
4 from leanrisk.models.gbm.explicit import moments as gbmm
5 from leanrisk.tests.custom_test_case import CustomTestCase
6
7
8 class TestGbmMomentsVsLogNormal(CustomTestCase):
9
10     @classmethod
11     def setUpClass(cls) -> None:
12         np.random.seed(1)
13         cls.num_params = 100
14         cls.x0 = np.random.uniform(low=0.1, high=200, size=cls.num_params)
15         cls.mu = np.random.uniform(low=-0.1, high=0.1, size=cls.num_params)
16         cls.sigma = np.random.uniform(low=0.01, high=0.3, size=cls.num_params)
17         cls.t = np.random.uniform(low=0.01, high=50, size=cls.num_params)
18
19     def test_mean_vs_lognormal(self):
20         for i in range(self.num_params):
21             with self.subTest():
22                 m = np.log(self.x0[i]) + (self.mu[i] - 0.5 * self.sigma[i] ** 2) * self.t[i]
23                 v = self.sigma[i] ** 2 * self.t[i]
```

## Example: Benchmark gone wrong after auto coding

test\_vs\_scipy.py

```
1 import numpy as np
2 from leanrisk.models.gbm._constant import moments as gbmm
3 from leanrisk.tests.custom_test_case import CustomTestCase
4
5
6 class TestGbmMomentsVsLogNormal(CustomTestCase):
7
8     @classmethod
9     def setUpClass(cls) -> None:
10         np.random.seed(1)
11         cls.num_params = 100
12         cls.x0 = np.random.uniform(low=0.1, high=200, size=cls.num_params)
13         cls.mu = np.random.uniform(low=-0.1, high=0.1, size=cls.num_params)
14         cls.sigma = np.random.uniform(low=0.01, high=0.3, size=cls.num_params)
15         cls.t = np.random.uniform(low=0.01, high=50, size=cls.num_params)
16
17     def test_mean_vs_lognormal(self):
18         for i in range(self.num_params):
19             with self.subTest():
20                 m = np.log(self.x0[i]) + (self.mu[i] - 0.5 * self.sigma[i] ** 2) * self.t[i]
21                 v = self.sigma[i] ** 2 * self.t[i]
22                 # mean of LogNormal(m, v) = exp(m + v/2)
23                 self.assertClose(
```

## 1 Intro

## 2 Agentic Quant Development

## 3 Validation Approaches

- Test Suites

- Mutation Testing

- Verification & Formalization

## 4 Conclusions

## Mutation Testing: Would the current test suite find a bug?

Mutation testing is a verification technique that tests if the test suite would detect code changes (mutants). It is performed in multiple steps:

1. Select a module of the code base you wish to verify.
2. Select a corresponding module of the test suite you use for testing.
3. Ensure the test suite is all green at the start.
4. Iterate over the code and introduce mutants, e.g. perform systematic mutations
  - replace `+` with `-`
  - replace `*` with `/`
  - ...
5. For each mutant, re-execute the test suite and check if it turns red (“mutant killed”) or green (“mutant survived”).
6. Surviving mutants are an indicator of weaknesses in testing.

Standard frameworks for this exist, e.g. `mutmut` in Python, and **agents can help in setting it up.**

## Mutation testing report

Killed 1970 out of 2667 mutants

| File                                                                     | Total | Skipped | Killed | % killed | Survived |
|--------------------------------------------------------------------------|-------|---------|--------|----------|----------|
| <a href="#">leanxva/mathematics/abm\constant\moments.py</a>              | 14    | 0       | 10     | 71.43    | 4        |
| <a href="#">leanxva/mathematics/abm\moments.py</a>                       | 74    | 0       | 55     | 74.32    | 19       |
| <a href="#">leanxva/mathematics/abm\simulation.py</a>                    | 17    | 0       | 17     | 100.00   | 0        |
| <a href="#">leanxva/mathematics/ajhw\auxiliary.py</a>                    | 363   | 0       | 237    | 65.29    | 126      |
| <a href="#">leanxva/mathematics/ajhw\moments.py</a>                      | 376   | 0       | 349    | 92.82    | 27       |
| <a href="#">leanxva/mathematics/ajhw\pricing.py</a>                      | 131   | 0       | 78     | 59.54    | 53       |
| <a href="#">leanxva/mathematics/ajhw\simulation.py</a>                   | 144   | 0       | 82     | 56.94    | 62       |
| <a href="#">leanxva/mathematics/black_scholes\constant\moments.py</a>    | 33    | 0       | 33     | 100.00   | 0        |
| <a href="#">leanxva/mathematics/black_scholes\constant\simulation.py</a> | 1     | 0       | 1      | 100.00   | 0        |
| <a href="#">leanxva/mathematics/black_scholes\simple\moments.py</a>      | 40    | 0       | 40     | 100.00   | 0        |
| <a href="#">leanxva/mathematics/black_scholes\simple\simulation.py</a>   | 21    | 0       | 0      | 0.00     | 21       |
| <a href="#">leanxva/mathematics/black_scholes\moments.py</a>             | 80    | 0       | 44     | 55.00    | 36       |
| <a href="#">leanxva/mathematics/black_scholes\pricing.py</a>             | 65    | 0       | 54     | 83.08    | 10       |
| <a href="#">leanxva/mathematics/black_scholes\sensitivities.py</a>       | 182   | 0       | 150    | 82.42    | 32       |
| <a href="#">leanxva/mathematics/black_scholes\simulation.py</a>          | 8     | 0       | 7      | 87.50    | 1        |
| <a href="#">leanxva/mathematics/correlation.py</a>                       | 27    | 0       | 17     | 62.96    | 10       |
| <a href="#">leanxva/mathematics/credit.py</a>                            | 5     | 0       | 5      | 100.00   | 0        |
| <a href="#">leanxva/mathematics/fx.py</a>                                | 5     | 0       | 5      | 100.00   | 0        |
| <a href="#">leanxva/mathematics/gaussian_model\pricing.py</a>            | 55    | 0       | 27     | 49.09    | 28       |

## Example:

consistency\_is\_not\_enough.py

```
1  from numpy.testing import assert_allclose
2  import numpy as np
3
4  def a(x):
5      return 2 * x
6
7  def b(x):
8      return a(x) + 1
9
10 # a change detector test for b would never kill a mutant on its own
11 # (tautological before any change) no matter how big the test instance
12 rng = np.random.default_rng(seed=1)
13 x = rng.uniform(low=-10.0, high=10.0, size=100000)
14 assert_allclose(b(x), a(x) + 1, rtol=1e-12, atol=0.0)
15
16 # a tiny regression test for b would kill the mutant
17 x = np.array([2, 3, 4])
18 bx = np.array([5, 7, 9])
19 assert_allclose(b(x), bx, rtol=1e-12, atol=0.0)
```

## 1 Intro

## 2 Agentic Quant Development

## 3 Validation Approaches

- Test Suites

- Mutation Testing

- Verification & Formalization

## 4 Conclusions

Quant code has two additional sources of error that go beyond software testing:

1. **Divergence between Methodology & Implementation:** Have we implemented the correct formula?  
**Agentic Practice:** Add the methodology documentation as code (e.g. `.tex`) into the repo and add a skill to check for consistency.
2. **Methodological Flaws:** Have we derived the correct formula?
  - 2.1 **Agentic Practice:** simply ask the agent to proofread the methodology document (perhaps add a skill and ask it to re-read after major changes)  
⇒ no guarantees, but can be quite helpful
  - 2.2 **Agentic Practice:** ask the agent to formalize the methodology documentation using, e.g., Lean or SymPy  
⇒ formal guarantees, but can be quite painful

# Example: Code <-> Methodology: Finding Quant Bugs

Good question to pin down, because the bug I just fixed was essentially a collision between two formulations. Here's the read:

What we simulate: the mean-zero factor (Brigo-Mercurio "x(t) +  $\phi(t)$ ")

The decisive evidence is conditional\_mean\_state:

```
return x_t0 * np.exp(-kappa * (t1 - t0))    # pure decay, no drift term
```

So the simulated factor obeys  $d\hat{x} = -\kappa \hat{x} dt + \sigma dW^Q$ ,  $\hat{x}(0)=0$ , hence  $E^Q[\hat{x}(t)] = 0$ ,  $\text{Var} = y(t) = \sigma^2/2\kappa \cdot (1-e^{-2\kappa t})$ . A mean-zero Gaussian factor is the signature of the Brigo-Mercurio deterministic-shift decomposition (a.k.a. Hull-White as extended Vasicek):  $r(t) = x(t) + \phi(t)$ , where  $x$  is the centered factor and  $\phi(t) = f(0,t) + (\sigma^2/2\kappa^2)(1-e^{-\kappa t})'$  is the deterministic curve-fit-plus-convexity. HullWhiteRate builds  $f_0 + \hat{x}$ , i.e. the forward curve plus the centered factor.

The other formulation – and why the bug happened

There's a competing, equally standard convention (Andersen-Piterbarg, "auxiliary y(t)":  $r(t) = f(0,t) + x(t)$  where the factor carries a y(t) drift –  $dx = (y(t) - \kappa x)dt + \sigma dW$  – so it is not mean-zero. In that world, the textbook bond reconstitution is exactly

$$P(t,T) = P(0,T)/P(0,t) \cdot \exp(-G \cdot x(t) - \frac{1}{2} G' \cdot y(t))$$

...which is precisely the  $-\frac{1}{2} y(t) \cdot G'$  term the old bond\_reconstitution used.

The bug was mixing them: the code simulated the mean-zero (Brigo-Mercurio) factor but reconstituted bonds with the Andersen-Piterbarg convexity term. The two formulations' factors differ by exactly the deterministic convexity  $(\sigma^2/2\kappa^2)(1-e^{-\kappa t})'$  – which equals  $\text{Cov}[\hat{x}(t), \hat{I}(t)]$  – so feeding the centered  $\hat{x}$  into the AP formula omits a  $-G \cdot \text{Cov}[\hat{x}, \hat{I}]$  shift. That's the term my fix restored, which is why everything became consistent (and why the discrepancy factor was exactly  $\exp(G \cdot \text{Cov}[\hat{x}, \hat{I}])$ ).

# Example: Lean

ABM.lean

```
1  import Mathlib.Probability.Distributions.Gaussian.Real
2
3  open ProbabilityTheory MeasureTheory
4
5  namespace Pyleanxva.ABM
6
7  /-- Law of the constant-parameter ABM at time `t`, started
8  at `x_0`:
9  `X_t ~ N(x_0 + μ·t, σ²·t)`. The variance parameter of
10  `gaussianReal` is a
11  `NNReal`, so `σ²·t` is inserted via `Real.toNNReal`. -/
12  noncomputable def abmLaw (x_0 μ σ t : ℝ) : Measure ℝ :=
13    gaussianReal (x_0 + μ * t) (Real.toNNReal (σ ^ 2 * t))
14
15  /-- (ABM-1) `E[X_t] = x_0 + μ·t`. The mean is the location
16  parameter of the
17  Gaussian; no sign condition is needed. -/
18  theorem abm_mean (x_0 μ σ t : ℝ) :
19    ∫ x, x ∂(abmLaw x_0 μ σ t) = x_0 + μ * t := by
20    rw [abmLaw]
21    exact integral_id_gaussianReal
22
23  /-- (ABM-2) `Var[X_t] = σ²·t`. The variance is the scale
24  parameter; `σ ≤ t`
25  lets the `NNReal` parameter coerce back to `σ²·t` on the
26  reals.
27  (If `variance_id_gaussianReal` is stated over `fun x => x`
28  in your Mathlib
29  revision, use `variance_fun_id_gaussianReal` instead.) -/
30  theorem abm_variance (x_0 μ σ t : ℝ) (ht : 0 ≤ t) :
31    variance id (abmLaw x_0 μ σ t) = σ ^ 2 * t := by
32    rw [abmLaw, variance_id_gaussianReal]
33    exact Real.coe_toNNReal (σ ^ 2 * t) (mul_nonneg (sq_nonneg
34    σ) ht)
35
36  /-- (ABM-3) `Std[X_t] = σ·√t`, the terminal object the
37  library exposes. Uses
38  `0 ≤ σ` and `0 ≤ t` to split the square root. -/
39  theorem abm_std (x_0 μ σ t : ℝ) (hσ : 0 ≤ σ) (ht : 0 ≤
40    t) :
41    Real.sqrt (variance id (abmLaw x_0 μ σ t)) = σ * Real.sqrt
42    t := by
43    rw [abm_variance x_0 μ σ t ht, Real.sqrt_mul (sq_nonneg σ)
44    t, Real.sqrt_sq hσ]
45
46  end Pyleanxva.ABM
```

## Have we implemented the correct formula?

**SymPy** is a Python-based computer algebra system capable of symbolic derivations that seamlessly integrates with numerical Python code. This enables an additional type of check to verify, e.g., a **derivative** of a function implemented in, say, NumPy:

- ▶ implement the function again in sympy,
- ▶ test sympy vs numpy consistency by evaluating both on a large set of randomly generated inputs,
- ▶ use symbolic differentiation in sympy,
- ▶ test the derivative in sympy vs numpy.

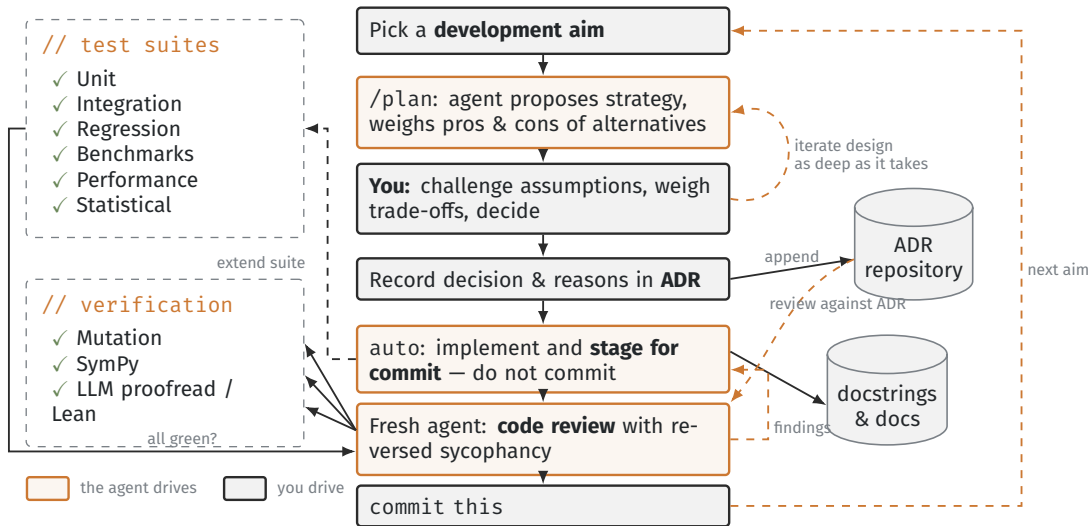
For **integrals**, one can use sympy or `scipy.integrate.quad`.

test\_vs\_sympy.py

```
1  from numpy.testing import assert_allclose
2  import numpy as np
3  import sympy as sp
4
5  # numpy implementations
6  def f(x):
7      return np.exp(x ** 2)
8
9  def df(x):
10     return 2 * x * np.exp(x ** 2)
11
12  # sympy implementations
13  x_sym = sp.Symbol('x', real=True)
14  f_sym = sp.exp(x_sym ** 2)
15  df_sym = sp.diff(f_sym, x_sym)
16
17  # lambdify converts sympy to numpy function
18  f_sym_num = sp.lambdify(x_sym, f_sym, 'numpy')
19  df_sym_num = sp.lambdify(x_sym, df_sym, 'numpy')
20
21  # perform test
22  rng = np.random.default_rng(seed=1)
23  x = rng.uniform(low=-3.0, high=3.0, size=1000)
24  assert_allclose(f(x), f_sym_num(x), rtol=1e-12, atol=0.0)
25  assert_allclose(df(x), df_sym_num(x), rtol=1e-12, atol=0.0)
```

- 1 Intro
- 2 Agentic Quant Development
- 3 Validation Approaches
- 4 Conclusions**

# Simple Agentic Workflow



## What I like about this flow

Human involvement is **gradual and controlled**:

- ▶ **Design stage**: level of detail to be handed to the agent and extent of sparring can be varied
- ▶ **Implementation stage**: plans are implemented fully but size and scope can be set and some parts can still be implemented by hand  
Highly effective variant: implement an example with a top-level API yourself
- ▶ **Human review** can be performed on multiple levels:
  - example use of code and review of outputs
  - code review of test suites in part or in entirety
  - code review of all library code

Once the library or environment gets bigger or more complex, the workflow can be enhanced with **multi-agent fan-out** (may happen automatically these days), **skills**, **hooks** or **MCP servers**.

## Humans remain indispensable for this workflow in order to...

- ▶ **specify the business requirements** the software needs to fulfill.
- ▶ **provide high quality input to the language models:** How do you write really good prompts? (“prompt engineering”) By knowing what you are doing.
- ▶ **enable understanding** of the solution as a whole.
- ▶ **provide oversight** of the workflow and the testing and verification framework.
- ▶ **be responsible and accountable** for the outcome.

## Regulation is very clear

*financial entities that have in place contractual arrangements for the use of ICT services to run their business operations shall, at all times, remain **fully responsible** for compliance with, and the discharge of, all obligations under this Regulation and applicable financial services law; DORA, Art. 28.a*

*The **responsibility** of the institutions' and payment institutions' management body for the institution or payment institution and all its activities **can never be outsourced**. EBA/GL/2019/02, §6*

*institutions should **maintain adequate in-house knowledge and core competence**, as they are ultimately responsible for outsourced tasks and functions, with enough access to **take direct control of an outsourced activity** in extremis, ECB Guide to Internal Models, §87*

## Upside Potential

- ▶ unleashes supercharged, passionate quants to build great systems and do research much quicker
- ▶ may change the effort estimates of big, exciting projects (e.g. AAD)
- ▶ may lower the cost of refactoring: opportunity to clean up old code
- ▶ acts as a powerful design assistant for more robust, maintainable code, if used consciously
- ▶ may lower the cost of creating test suites
- ▶ professionalizes the software design process
- ▶ reduces the time to find stupid mistakes
- ▶ may help to keep code and documentation in sync

## Downside Risks

- ▶ very easy to quickly generate huge amounts of AI slop (including test suites)
- ▶ flooding human experts with demands to review AI-generated content
- ▶ may create false sense of certainty in model outputs
- ▶ could lead to loss of human oversight
- ▶ can create difficult questions of accountability, governance and oversight
- ▶ may lead to reduced human ability in mathematics and programming in the long run
- ▶ can create unrealistic management expectations