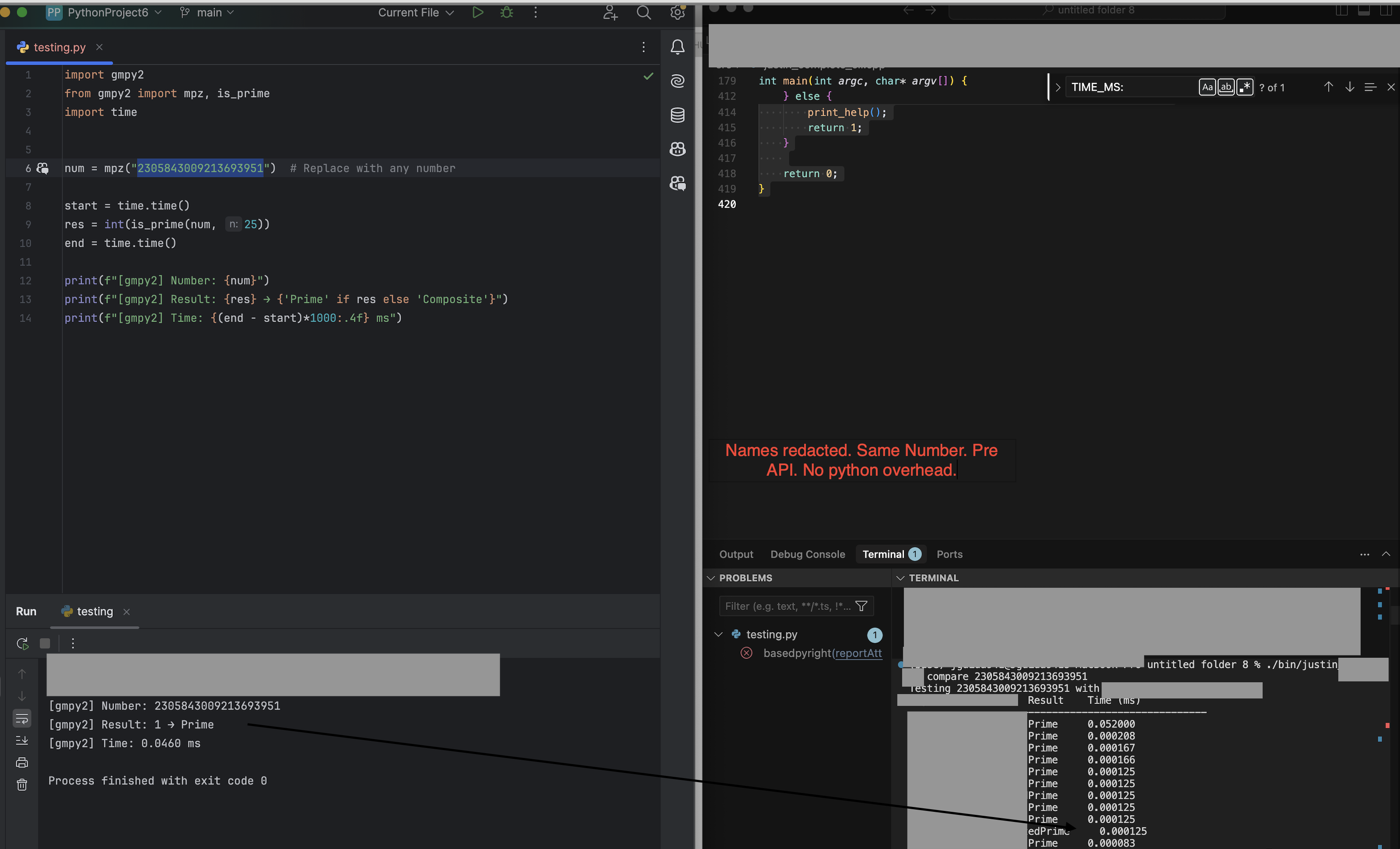
**My Beliefs: System Design Process Models**

**Justin Guida — Message or contact me if you’re interested in collaboration. I can’t do all this myself.**

I mainly work in Python and C++, mostly building layered systems like primality tests, password-cracking systems, and my prize: a platform I’m evolving into a callable API called SmartAuto.

Think is\_prime()  but smarter, faster, and fully transparent, no black box AI I will leave a picture with names marked out for privacy for you guys to see my calls against gympy2. I have a cli so I still need an api to have it be a fair run through python cause overhead but the numbers are why this has been 3 months in the making.

Pretty beautiful

What’s beautiful is that my code is pure C++, not some wrapper glued to Python. No cheap tricks. I built on GMP’s architecture and beat it.  Which is kinda comical if you know how optimized GMP already is.

These aren’t just scripts  they’re architected, benchmarked systems grounded in math. I’ve gone head-to-head against SymPy, GMP, GMPY2, NumPy, and more  and won. And to push past GMP’s performance ceiling, I even hand-coded SIMD NEON routines on my M4 Mac.  Architecture and design is everything and that picture should be more then proof.

That kind of speed doesn’t happen by accident. It takes structure.

**Why Process Models Matter To Me:**

They’re not just a class assignment .... they’re the only thing keeping your system from collapsing when one part changes. You have to deeply understand that. Especially in C++, where you’re juggling low-level memory, SIMD, and 40+ source files, design isn’t optional.

I’ve skipped it before and it always ends the same: lost hours, broken logic, and rewrites and a headache.

Now I rely on structure. I use physical notebooks, mermaid type flowcharts, architecture maps, file plans. So many .mds. Not because it’s old-school, but because I can flip to a page faster than I can grep through a codebase. For me, documentation is part of the system not an afterthought, it's simply necessary in high end systems, and should be the goal (at-least in my opinion) of everything you code for it to be cleanly written, optimized the best, good comments with proper architecture fro the hardware and software side.

A good architecture isn’t just a UML diagram  it’s a living strategy for how components evolve, interact, and stay stable. The more I've learned about programming the more I've learned architecture is the most important aspect. If you plan like slop that's exactly what your gonna get.

**Design ≠ Optional**

Even a rough flowchart can tame a monster system,. I used to think diagrams were for beginners, and I was wrong.

You want to build a pure math, SIMD-accelerated primality tester with 0.0008s latency?

You start with architecture. Only then can you optimize with confidence .I call it the flow state!

**Importance of Consistency (DFD Guidelines)**

When it comes to **Data Flow Diagrams (DFDs)**, consistency is king. Consistency is a rule I follow more than anything else.

In one of my benchmarking suites, a single inconsistency for a edge cases **broke the entire output**. It wasn’t a bug it was a **design failure on my part, but also a lesson, one of many!**

Once I standardized every layer’s logic, everything lined up. Suddenly I could optimize **deeply**. That’s when the system **flew**.

**How I’d Model  Project One**

Same way I model any advanced system:

**Understand the Requirements:**

Figure out what’s being built

If it’s an app, who’s using it?

If it’s a mathematical engine: like FFT, SAT solving, or primality detection , I reverse-engineer what’s out there and being used as the gold standard.

Then I find the math, the gaps, the opportunity and learn.

**Build the Core Logic**

Start with a flowchart or sketch. Then build modular code.

Always leave comments: especially around advanced math... not just for others, but for my future self!

Thousands of lines later, that context is everything.

**Validate the Output:**

Most of my time is in **testing and benchmarking**.

**I break things on purpose.**

Then I iterate.

I’ve built over **100 versions** of my system in Python and C++, each tuned for a different purpose.

Right now, I’m learning **pybind11** to expose my C++ core to Python — easily the most complex API I’ve ever written.

This is where your architecture either **holds up… or collapses**. It's stressful

**My Rules for Every Project**

Clean code with deep comments (even if some say i'm “too detailed”)

Modular everything, unless prototyping

NEVER AGAIN: will I write an 8,000-line file (shoutout to my first PyQt6 project )

Git as a religion; always backed up, always versioned

Flowcharts always : to teach myself and others

And most importantly: A design that survives pressure

**Final Thoughts:**

This class finally gave structure and vocabulary to what I’ve learned the hard way:

Strong architecture isn’t optional.

It’s the only thing keeping your system from falling apart as it scales, breaks, or evolves.

And once you’ve built something that outperforms GMP…

You understand exactly how important that architecture really is.

— Justin Guida