The LLM-Era Programmer's Toolkit

A reflection on expertise, tools, and how AI changes the calculus

---

The Premise

It's very hard to be on top of all the most powerful tools in programming.

Most programmers have domains of competence.

There are diminishing returns for becoming an expert in all tools because life is finite.

It's logical for a programmer to solve things within their domain of competence.

LLMs are at least competent, sometimes expert, in all tools of programming.

Therefore, in the new era, it makes sense for programmers to have a general awareness of all the most powerful tools and to lean on the AI for the expertise.

I'm a proponent of functional programming, especially PureScript and Haskell. However, I know there are a bunch of technologies that are also incredibly powerful that I don't use because my expertise is limited.

I'd like to make a list to keep in mind of things that I probably should reach for—things I previously wouldn't have reached for because of yak-shaving and learning curves—that I should now consider because I have Claude to guide me.

Here's my starting list of things I'd like to level-up with if I had time/brain-cells:

- Program proving: TLA+/Alloy, Liquid Haskell, Agda, Coq
- Logic programming: SMT solvers, Prolog
- Database design and management
- Deployment: containers, Kubernetes, AWS etc.
- Other visualisation libraries: R, ggplot
- LLM tuning

What am I missing, both in those categories and in entirely different fields?

The Expanded List

Additions to the starting categories

Program proving: Add **Lean 4** (much better ergonomics than Coq, very active community), **Dafny** (verification baked into a usable language), **F*** (Microsoft's verified programming language).

Logic programming: Add **Datalog** (Souffle, DDlog)—transformative for program analysis, static analysis tools, and surprisingly many problems. Also **Answer Set Programming** (clingo) and **constraint programming** (MiniZinc).

Database: This category deserves expansion:

- **DuckDB** — embedded OLAP, absurdly fast for analytics
- **Vector databases** (Qdrant, Weaviate) — essential for LLM/RAG work
- **Graph databases** (Neo4j, or Datalog-based like Datomic)
- **SQLite as serious infrastructure** (not a toy—see Litestream, LiteFS)

Deployment: Add **Nix/NixOS** — transformative and aligns with FP thinking. Also **Terraform/OpenTofu** for Infrastructure as Code.

Categories missing entirely

1. Probabilistic programming

Stan, PyMC, Turing.jl, Gen. Bayesian inference as a programming paradigm. If you have an FP background, this should feel natural—it's essentially "functional programming for uncertainty."

2. Systems/performance programming

- **Rust** — you probably know of it but may not reach for it
- **SIMD/vectorization** — knowing when to use it
- **GPU compute** (CUDA concepts, but also **WebGPU** which is browser-accessible)
- **Memory-mapped I/O, zero-copy patterns**

3. Streaming and event architectures

- **Kafka/Pulsar** for event streaming
- **Event sourcing / CQRS** patterns
- **CRDTs** for distributed state (connects naturally to FP interests)

4. Fuzzing and property-based testing at scale

Beyond QuickCheck:

- **AFL/libFuzzer** for finding bugs in native code
- **Hypothesis** (Python) for more sophisticated property testing
- **Mutation testing** for test quality assessment

5. Zero-knowledge proofs

ZK-SNARKs/STARKs are becoming mainstream infrastructure, not just crypto curiosities. Tools like **Circom**, **Noir**, **Halo2**.

6. Observability stack

- **OpenTelemetry** (distributed tracing)
- Structured logging and log aggregation
- Metrics pipelines (Prometheus/Grafana)

7. Parser infrastructure

- **Tree-sitter** — the modern way to parse code for tooling
- **LSP** (Language Server Protocol) — for building dev tools

8. Symbolic computation

- **Computer algebra systems** (SymPy, or Mathematica concepts)
- **Automatic differentiation** (JAX, but also understanding the technique)

9. Simulation paradigms

- **Discrete event simulation**
- **Agent-based modeling** (Mesa, NetLogo concepts)
- **Monte Carlo methods**

10. Build reproducibility

Nix deserves its own category. Content-addressable builds change how you think about deployment entirely.

The Meta-List

A mental model for the LLM-era programmer's toolkit:

Category	What to know exists	When to reach for it
Correctness	Lean, TLA+, property testing, fuzzing	When bugs are expensive
Performance	Rust, SIMD, GPU, profiling	When it matters (measure first)
Scale	Kafka, k8s, distributed consensus	When single-machine breaks
Data	DuckDB, vector DBs, graph DBs	Match structure to query pattern
Uncertainty	Probabilistic programming, Bayesian	When you have beliefs + data
Search	SMT, constraint programming, SAT	When brute force is too slow
Reproducibility	Nix, containers, IaC	When "works on my machine" hurts
Observability	Tracing, metrics, structured logs	Before you need to debug prod

---

The Shift

The LLM shift means the **learning curve cost has collapsed**, but the **awareness cost hasn't**.

The new model: maintain the map, outsource the territory navigation.

You don't need to be an expert in Kubernetes to deploy to Kubernetes effectively. You don't need to memorize TLA+ syntax to write a specification that catches a concurrency bug. You don't need to know the DuckDB API to run a complex analytical query.

What you *do* need is:

1. Awareness that these tools exist
2. Understanding of what problems they solve
3. Judgment about when to reach for them

The AI provides the expertise on demand. You provide the taste and the judgment.

January 2026