

The More Code, The Faster You Go: How a Healthy Codebase Accelerates Development

Introduction

In the software world, it's almost taken as a law of nature that *more code equals slower progress*. Large codebases often become tangled with technical debt, making every new feature a slog. However, our experience as programmers has revealed an exciting counterpoint: **with a healthy codebase and good practices, the more code you have, the faster you can go**. This is a paradigm we rarely discuss, yet it's a visible marker of good software design. In a well-architected project with robust tests, clean abstractions, and smart reuse, each additional component becomes an asset that **speeds up** development rather than slowing it down. This article, co-authored by both of us, explores how building **reusable components** and investing in quality flips the script on the typical "big code = slow progress" narrative. We'll illustrate this through our journey of commoditizing code elements, leveraging Wardley Mapping concepts, and reaping the rewards in development velocity.

Innovate, Leverage, Commoditize: The Wardley Maps Approach

One way to understand this phenomenon is through the lens of **Wardley Maps** and the *ILC pattern* – **Innovate, Leverage, Commoditize**. Simon Wardley describes a strategy where you *innovate* something new, *leverage* it by refining and scaling it, and then *commoditize* it into a utility or product that others can easily reuse ¹. In practice, this means **taking a custom-built solution and turning it into a standardized component**. Once a piece of technology becomes a commodity or utility, it **frees you to build higher-order innovations on top of it**. As Wardley puts it, you "take something which is a product, turn it into a utility, allow everybody else to build on top of it... and that way, you just move up the stack" ¹. Each cycle of commoditization paves the way for new creativity at the next level.

In our coding journey, we applied this ILC cycle at the codebase level. Whenever we developed a useful bit of functionality, we invested effort to **abstract it, stabilize it, and package it as a reusable service or module**. This strategy might sound time-consuming, but it's actually a *sensing mechanism* for opportunities and a force-multiplier for speed. Every time we solidified a component (turning it from a one-off script into a robust library or service), we noticed that it enabled faster development of features that depended on that component. It's the software equivalent of paving a road so you can drive faster next time, instead of bushwhacking anew for each trip.

Building Blocks: From Custom Code to Reusable Services

Let's illustrate how this works with a concrete example from our projects. Early on, we found ourselves writing a lot of code to handle file storage in different environments (local disk, cloud storage, archives, databases, etc.). To simplify this, **we built a `MemoryFS` – an in-memory file system abstraction layer**. This `MemoryFS` component allowed our code to save and retrieve files without caring *where* the files were actually stored. Whether the data lived on the local filesystem, an Amazon S3 bucket, a ZIP archive, or a MySQL database, our application code didn't need to know. We commoditized file storage into a self-contained module.

On top of this foundation, we then built a **caching service**. Since `MemoryFS` made storage location irrelevant to the application, the caching service could introduce powerful strategies transparently. For instance, our caching service can save a file in multiple ways with one call – say, storing one copy in a "latest version" location and another in an archival or versioned folder. With this approach, every time we save data through the cache, we automatically get a current version (for quick access) *and* a historical backup. The application using the cache doesn't have to implement any of that logic – it's all handled by the commoditized caching component. We further exposed this caching functionality via a **web service API** (built with FastAPI), and provided a lightweight **client library** (in Python) for any other tool to integrate with the caching service effortlessly.

This layering continued: for example, an HTML generation service we created was able to use the caching *client library* directly – meaning it could cache its outputs simply by calling our API client, without knowing anything about how caching or file storage works internally. Behind the scenes, the client talked to the caching web service, which used `MemoryFS`, which in turn used either local disk or cloud storage as needed. Each layer abstracted the complexity of the layer below it.

Crucially, we also **productized our deployment and infrastructure code**. We developed a *base service template* that had CI/CD pipelines, testing frameworks, and deployment scripts baked in. When we start a new service now, we begin with this template – which means **from day one, the service can build, test, and deploy (to dev, QA, or production) without adding any custom deployment code**. For example, our FastAPI-based services can be deployed as serverless functions with all the plumbing already handled by the base template. This was another custom solution we commoditized: deployment itself became a reusable component.

The result? Each new piece of code we add to the system is built on top of a growing tower of well-engineered components. We're not reinventing file handling, caching, or deployment for each project – we **reuse and leverage** the work we've already done. In a very real sense, we have more code in our codebase than ever, but that code is organized into layers of capability. We can now assemble new applications like snapping together LEGO bricks, rather than carving each piece from raw wood. This is the essence of modular design: *compose* new systems from battle-tested modules instead of coding everything from scratch. And composition scales far better than construction ² – meaning our capacity to build accelerates as our library of components grows.

More Code, More Speed – The Positive Feedback Loop

With this approach, we observed a powerful positive feedback loop: **the more code we accumulated (in the form of reusable modules), the faster we could deliver new features**. This is the opposite of the typical scenario where more code means more bloat and drag. Why were we getting faster? There are several reasons:

- **Higher-Level Focus:** Because lower-level problems were solved once and for all, we could focus on higher-level innovation. Every time we commoditized a functionality (like file storage or caching), it cleared cognitive overhead for the next projects. New ideas could build on *existing* foundations instead of starting from ground zero. In Wardley Mapping terms, each commoditized component enabled a new *genesis* of ideas above it ¹.
- **Reduced Friction:** Development friction dropped significantly. Need to add caching to a feature? We already have a caching service – just plug it in. Want to deploy a new microservice? Use the base template – no need to set up pipelines from scratch. The workflows became streamlined

and standardized. With such reduced friction and pre-built infrastructure, *developers can ship features faster and more reliably* ³.

- **Parallel Development:** Different team members (or different services) can work in parallel without tripping over the same problems. One person uses the file system module while another uses the caching API; both can proceed without needing to solve each other's problems. The standardized interfaces between modules mean progress doesn't require tight coordination across the entire stack. We effectively **decoupled development efforts** by having clear module boundaries.
- **Confidence and Stability:** A well-tested, modular codebase gives a lot of confidence. When you know your base components are solid (thanks to thorough automated tests), you can build boldly on top of them. If something goes wrong in a new feature, it's likely isolated to the new code, not a mysterious bug in a foundation class. This stability means we spend less time debugging old code and more time writing new code. It creates a virtuous cycle of speed: quick releases, followed by quick feedback, leading to even quicker improvements. In fact, industry data shows that high performers can achieve both **speed and stability** – there's no inherent trade-off between moving fast and maintaining quality when engineering practices are sound ⁴ ¹.

In short, every piece of code we added was a force multiplier for future development. Instead of adding complexity, our additions *added capability*. Our codebase became richer, not more complicated – because we continuously refined the design and kept complexity encapsulated in the right places. This is how **a well-designed system turns code volume into an advantage**. It's akin to an ecosystem: the more robust the roots and trunk, the more branches and fruits you can support.

Why Bad Code Usually Slows You Down

It's important to contrast this with the more common situation. Most developers have felt the pain of a growing, messy codebase where each additional line seems to make everything creak. **Technical debt** is the usual suspect. Technical debt represents all the quick-and-dirty solutions, outdated hacks, and poor-quality code that remain in the system. Such code might "work," but it's of *sufficiently poor quality that it causes us to move slower when implementing new functionality* ⁵. If every new feature requires wading through muddy code or rewriting brittle components, progress grinds to a halt.

Studies have quantified this drag. For example, a 2018 study found that *developers waste on average 23% of their working time due to technical debt* ⁶. That's basically two hours out of every eight-hour day lost to wrestling with legacy issues. Even worse, unchecked technical debt tends to snowball: nearly a quarter of the time when developers encounter existing debt, they are forced to introduce **additional** hacks or workarounds, compounding the problem ⁶. This creates a downward spiral – the more code you have (in a bad state), the slower you get. As one author noted, if a team isn't regularly paying down debt, *the further we fall behind, the slower we go* ⁷. Eventually, developers start dreading adding new features because touching the code is so fraught with delay and risk.

In such an unhealthy environment, it's no surprise that "more code" correlates with "more mess." Without deliberate design, *complexity grows faster than functionality*, and your velocity tanks. Large enterprises often feel this acutely: years of accumulated code across hundreds of applications can become an anchor that drags down innovation speed. The difference between this scenario and the positive one we described earlier boils down to **design, architecture, and discipline**. Where a healthy codebase has clear abstractions and strong foundations, an unhealthy one has tangled dependencies

and fragile hacks. Where the former has modules you can build on, the latter has landmines you must tiptoe around.

Investing in Quality to Go Faster

So how do you ensure that more code makes you faster, not slower? The answer lies in investing in **non-functional requirements** and **engineering excellence** up front. This includes things like comprehensive testing, continuous integration pipelines, code review, refactoring, documentation, and modular architecture. These are often called "non-functional" aspects of a system (because they don't deliver a feature directly to the end-user), but in truth, they are the *bedrock* of sustained speed and agility.

In our case, having a strong test suite for each component meant we could confidently reuse those components in new contexts. Continuous integration and deployment (CI/CD) automation meant every service we built benefited from quick feedback and consistent releases. We treated internal infrastructure code (like our deployment templates and dev environment setup) as first-class citizens, worthy of the same care as user-facing features. This effort upfront pays dividends when scaling the codebase.

The Agile Manifesto, which guides modern software development, famously states: *"Continuous attention to technical excellence and good design enhances agility."* ⁴ In other words, **the best way to go fast is to go well**. Our experience confirms this 100%. By keeping quality high and architecture clean, we avoided the quagmire that slows teams down. Every hour spent writing tests or improving an interface was an investment in future speed. Rather than viewing refactoring or writing infrastructure code as drudgery, we saw it as building *road infrastructure* for our project: once the road is there, you can drive unbelievably fast on it. Skipping those investments is like driving off-road – it might feel quick at first to cut across the field, but soon you'll be stuck in the mud.

Another way to look at it is through the concept of **developer experience (DevEx)**. A healthy codebase creates a great developer experience – things work as expected, deployments are push-button, and the environment guards against errors. Companies with top-tier DevEx often use internal platforms or shared services to give developers high leverage. When workflows are streamlined and common problems solved centrally, developers can focus on delivering value. It's been observed that **streamlined workflows and reduced friction allow teams to ship features faster and more reliably** ³. This matches our outcome: by standardizing and commoditizing the gritty stuff (file I/O, caching, deployments, etc.), we reduced friction for every new piece of work.

In short, **investing in testing, refactoring, and modular design is not a tax on productivity – it is a powerful accelerator**. It flips the dynamic so that code accrues **assets** (reusable modules, stable interfaces, automated processes) instead of liabilities. This is why forward-thinking teams allocate time for maintaining and improving the internals of their systems. Far from slowing you down, that work ensures you can keep up a high development pace even as the codebase grows.

Conclusion

The idea that *"the more code you have, the faster you can go"* might sound counterintuitive in an industry used to fighting bloated legacy systems. But a key insight from our journey is that **code volume alone isn't the culprit – it's the nature of that code that matters**. When you cultivate a codebase with strong foundations, clear abstractions, and a culture of continually turning custom work into commodities, you unleash a flywheel of productivity. Each new module or service doesn't add drag; it

adds fuel to the engine. You create a landscape where higher layers of innovation blossom because the lower layers are rock-solid and accessible.

This is the promise of good software engineering practices: that as a system evolves, it actually gains momentum. Metrics and anecdotes align on this point – high-quality code and infrastructure lead to higher speeds, while poor-quality leads to decay and slowdown ⁶ ⁷. The visible indicator of good design is a team that *accelerates* over time, turning out features quickly without the wheels coming off.

Achieving this isn't magic. It requires upfront work, deliberate architecture, and sometimes the patience to slow down and fix things properly when you'd rather rush a quick hack. But as we've seen, those investments create a code ecosystem where the **ILC cycle** keeps spinning: innovate on top, leverage what's there, commoditize the new bits, and repeat. With each turn of the cycle, you climb higher and go faster.

In a world where technology changes rapidly, having a codebase that lets you respond rapidly is a huge competitive advantage. It means you can try bold ideas (since your robust platform has your back), and you can sustain a high tempo without burning out in a tar pit of debt. So the next time someone equates more code with more problems, remember that it's only true for unmanaged code. In a well-tended garden of code, growth leads to even more growth. **The more quality code you cultivate, the faster you can harvest new innovation.** And that is a paradigm worth talking about.

Sources: Continuous attention to technical excellence (Agile Manifesto) ⁴; Tech debt wastes ~23% dev time ⁶ and makes teams slower ⁷; Wardley on Innovate-Leverage-Commoditize cycle enabling higher-order innovation ¹; Streamlined platforms boost delivery velocity ³; Modular composition scales better than building from scratch ².

¹ Stuff The Internet Says On Scalability For October 4th, 2021 - High Scalability -

<https://highscalability.com/stuff-the-internet-says-on-scalability-for-october-4th-2021/>

² The Rise of Modular Development: Building Tech That Builds Itself - DEV Community

<https://dev.to/jaideepparashar/the-rise-of-modular-development-building-tech-that-builds-itself-30p8>

³ Internal Developer Platform [Benefits + Best Practices] | Atlassian

<https://www.atlassian.com/developer-experience/internal-developer-platform>

⁴ ⁵ ⁶ ⁷ Technical debt and productivity | Agile Technical Excellence

<https://agiletechnicalexcellence.com/2024/01/21/technical-debt.html>