TALKS EVERGREEN

Home-Cooked Software and Barefoot Developers

The emerging golden age of home-cooked software, barefoot developers, and why the local-first community should help build it

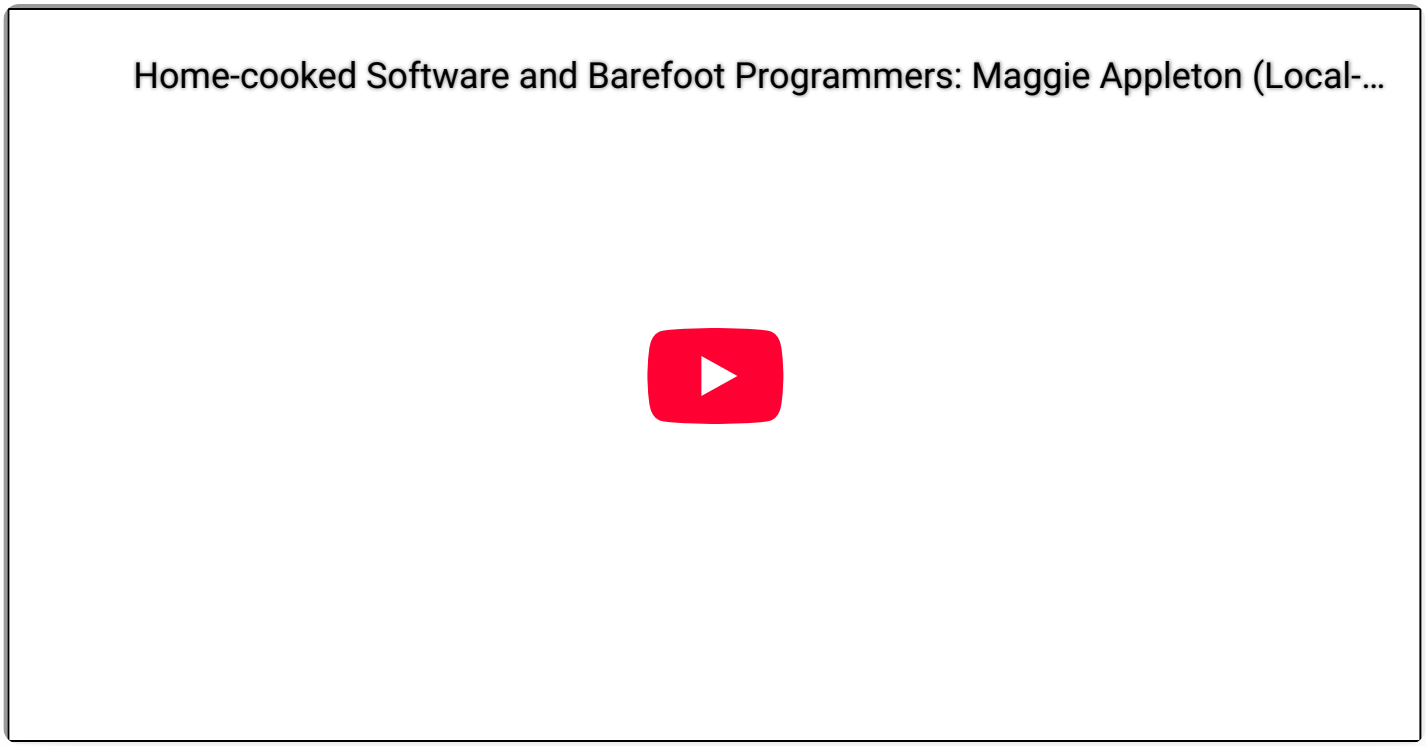
Anthropology Artificial Intelligence End-User Programming Language Models Tools for Thought

Planted 9 months ago
Last tended 8 months ago

This is a talk I presented Local-first Conference in Berlin, May 2024 1ya. It's specifically directed at the local-first community, but its relevant to anyone involved in building software.

For the last ~year I've been keeping a close eye on how language models capabilities meaningfully change the speed, ease, and accessibility of software development. The slightly bold theory I put forward in this talk is that we're on a verge of a golden age of local, home-cooked software and a new kind of developer – what I've called the barefoot developer.

Local-first Conference Video



Slides and Transcript

First, a quick intro. I'm Maggie. I look like this on the internet. I'm a product designer at a start-up called Elicit . We use machine learning and language models to make tools for scientific researchers. I'm also a

mediocre developer, meaning I try to build things but they sometimes don't work.

I write about and research lots of things in public, online including tools for thought, language model interfaces, and end-user programming.

Some of which I want to talk to you about today.

I'm here to convince you of two things.

First, that language models will create a golden age of local, home-cooked software and barefoot developers.

I'll explain these terms in a minute.

And the second thing I want to convince you of is that the local-first community has a significant role to play in building that future golden age.

And so here's what we're going to talk about:

Local-first Beyond Local Data

Home Cooked vs Industrial Software

My Pitch for Barefoot Developers

Why Language Model Legos Need Glue

How We Can Bake Local-first Into Everything.

So let's talk about local-first, beyond local data.

I know everyone here knows what local-first is about, but here's a quick recap for people watching who might not.

When we talk about "local-first," we're primarily talking about where data lives and how it syncs to our devices.

So in the old school world, before the cloud, our data and our software lived on a single machine and we had no way to collaborate. Except by throwing LAN parties.

We then moved to the cloud era where all the programs and all the data live in the cloud and stream to our devices via the internet.

And now, everyone here is trying to make a new, beautiful local-first future happen, where the software and the data all live on our local machines and just use the cloud to sync between them whenever we get internet access.

The existing philosophy of local-first is fantastic. I'm on board with it, I know you all are too.

But come expand your minds with me for a moment, and imagine a more holistic vision for local software.

What if it was about more than just databases?

Local traditionally means something close to your home, something familiar, intimate, and trusted by you, something unique to its location, and something community-oriented, as in supporting local businesses.

So there's an interpretation of local software that means software that is built close to the home and serves the needs of the home.

It's software someone might build for themselves, their family and friends, their neighborhood and community.

It solves local problems for local people.

This idea has been floating around with various names attached to it for the last 20 years.

Clay Shirky first proposed the concept of ' situated software ' in 2004 21ya .

He defined this as software “designed in and for a particular social situation or context.”

He painted a vision of applications that could be used by dozens of users rather than thousands or millions. This is an absurd target population both then and now.

The dream was that communities would get very form-fit tools for their particular needs, rather than trying to adapt generic software to solve them.

Unfortunately, Shirky was ahead of his time – it was practically impossible to build this small-scale software in 2004 21ya .

Sixteen years later, in 2020 5ya , Robin Sloan published a blog post called " An App Can Be a Home-Cooked Meal " which picked up on many of the same themes.

He talked about building a tiny app for his family to send short videos to one another. Only his family have access. He's not going to turn it into a start-up. It doesn't have any commercial or market value.

He made it out of care and love for the people around him to help them solve a small problem and have enjoyable experiences together.

Just like he would if he had made them a home-cooked meal.

Home-cooked apps, like meals, are apps you make for the people you know and love.

We've progressed enough since 2004 21ya that someone can make one of these on their own.

They're simple – they require some skills, but not tons.

They're cheap or free to run – you're not at the mercy of a corporate pricing strategy.

You control what happens to them – you control whether features are added or removed, and whether they'll be shut down or sunset.

Private – won't sell your data.

Serve your specific needs.

No financial pressure to monetize them or make a profit off them.

Safe – very little risk of nefarious actors or misaligned incentives.

Made with love and care.

I like collecting examples of these.

Here's an app to track newborn breastfeeding and diaper changes.

Why put your baby's bodily functions on someone else's server?

Here's one for reconciling personal finances that's described as a mashup between Superhuman and Tinder, which sounds much more fun than standard financial software.

A live sales dashboard made with an old phone - a nice way to reuse hardware.

This is one someone made for their diabetic partner. The default interface on their glucose monitor wasn't giving them the critical information they needed, so they took matters into their own hands and made one that did.

I think it's fitting that all of these examples of home-cooked apps deal with the kind of data you don't want leaving your personal home, like your children, bodily functions, health, and finances.

Robin Sloan's post on home-cooked apps hit a nerve. It's been a wildly popular piece endlessly passed around the programming community.

I think this is because it pointed to many tensions we have around the way software is currently built.

These small-scale, personal apps...

..are drastically different from the kind of software most of us use every day, which is professionally made software.

This is like a meal or cake made by a professional chef at a restaurant. Probably much more delicious than yours.

But it is likely more complex, more expensive to make, and requires many people with expertise.

You can't control how it's built, how long it lives for, or how much butter they put in it.

Has commercial interests at heart, rather than familial love and care.

Less security and privacy – you don't know where the data is going.

And professional software is the kind of software almost everyone in the world uses. It's made for masses of people, at a global scale, by huge corporations.

At the moment, we're in the industrial, high-modernism age of software, where these standardized, one-size-fits-all apps are made for us by people who don't know much about us.

Primarily people working for large Californian corporations and shipping their software overseas.

They're sitting in San Fransisco trying to understand our pains and problems over Zoom calls and customer support tickets.

They have so little context on our lives, what problems we need solved, and what we value.

How could an American getting paid six figures in Mountain View understand how to identify problems and design solutions for a

homemaker in Tokyo, a street seller in Turkey, or a doctor in Tunisia?

For the most part, they don't. Or if they try, they do it badly.

My friend Kasey Klimes wrote a fantastic piece called " When to Design for Emergence " on the design dynamics of large-scale software after working on Google Maps.

He points out that our current approach is designed to only solve the most common needs of the most number of users.

Anything beyond that is what we call the long tail of user needs. These are things only a few people need, but there's a nearly infinite amount of them.

In the case of Google Maps, the long tail of user needs is anything beyond “how do I get from here to there?”

Google Maps is never going to support showing historical borders or tidal patterns, even if those things are essential to a few dozen, or even a few hundred people.

In the current system, this stuff is always considered out of scope.

Because it doesn't make any financial sense to support the long tail.

Industrial software can only target the biggest problems for the most people, ideally wealthy people with disposable income.

This is an economic limitation. Building features that solve every single long tail need requires a lot of engineering labor...

...and engineers are very expensive.

For industrial software, you need large teams of developers and designers and product people who demand high salaries in exchange for their valuable, hard-won expertise.

And the way we fund this is primarily through US-based venture capital funding, which demands hockey stick growth in return.

All the focus goes into making hundreds of millions or ideally billions of dollars in profit to pay back their investors.

As a member of a team that's in the middle of doing this, I can say it affects every single decision about how we build the software and what features are prioritized.

I've drawn a little map to help make these dynamics clear.

We have scale on the horizontal axis, going from small, special snowflake software up to global industrial software.

And profit on the vertical axis, so unicorn companies making billions of dollars versus financial sinkholes.

Most software today sits in this upper right quadrant of being large-scale and aiming to make as much money as possible.

Anything that falls below the profit line ends up being a short-lived failure—it's not allowed to survive long in the marketplace.

But over here on the left-hand side of small-scale, specific software, we have the land of opportunity.

This kind of software doesn't even need to make a profit; it just has to operate at a very low cost, although maybe it could make some small to medium amount of profit over the long term.

But this land of opportunity currently has a skill issue.

Because the Venn diagram of people with the skills to make home-cooked software and professional developers is essentially...

...a concentric circle.

All the examples of home-cooked software I showed were made by people who work as professional developers.

The vast majority of people who might want to make a piece of home-cooked software can't because software development is too complex. Even those of us who are professionals don't always have the right knowledge to make a full-stack app for ourselves.

This isn't news to anyone. This has been the main complaint of end-user programming advocates for decades.

So widespread, local, home-cooked software is still just a pipe dream.

This is a shame because we know the world is full of problems to be solved. While not all of those problems have software-shaped solutions, a whole bunch of them do.

It's hard to argue it wouldn't be overwhelmingly net good for more people to be capable of designing and building their own software to solve problems for their local communities.

So what happens if some parts of software development get faster, easier, and cheaper?

Well, 4 years ago, OpenAI released GPT-3, the first meaningfully capable large language model. And since then, we've been on a bit of a ride.

Now, when I say large language model...

I'm talking about what everyone else calls AI. But I think that term is too general.

I'm specifically talking about models that are made using deep learning and neural networks.

These are primarily large language models, but this also includes vision and action models.

They are models that can understand words, code syntax, images, and interface actions based on human training data.

I'm also talking about what's come to be called agents.

This is when we get large language models to behave like an agent that can make plans and decisions to try and achieve goals we give them.

We give these agents access to external tools like web search, calculators, and the ability to write and run code.

As well as long-term memory stores in databases.

And we get them to mimic logical thought patterns like having them observe what they know, plan what they want to do next, critique their own work, and think step by step.

The agent gets to decide what tool it wants to use at any one point to solve the problem we've given it.

This architecture of chaining together tools and logic makes language models far more capable than they would be otherwise.

They end up being able to do quite sophisticated tasks within our existing programming environments.

Unless anyone here has been living under a rock, you know we've been deploying language models and agents into tools designed to help professional developers like GitHub Copilot, Cursor, and Replit.

They can read and write code, debug things, create documentation, and write tests.

One study showed that developers using Copilot were 55% faster at completing tasks, so we at least know this speeds people up.

I can say from personal experience I am a much better programmer with access to these tools, but I appreciate there's a lot of skepticism and controversy over them.

Perhaps they're just creating more crap code and bugs for everyone to deal with down the line.

But I actually don't want to talk about how advances in AI will affect professional developers. With all due respect, we have it pretty good.

Our problems are boring.

I want to talk about a different kind of developer.

What I call barefoot developers.

This is a term I've made up, based on the idea of...

Barefoot doctors.

Initiative by Mao's government in China in the 1960s. I'm not saying *everything* Mao did was great, but this was a pretty good program. I'm

sure he had very little to do with it.

At the time, 80% of the population lived in rural areas but had no or poor access to healthcare. Medical expertise was all concentrated in urban areas.

They selected people from rural villages and trained them up to become healthcare providers for their communities. These villagers were taught preventative care, curing simple ailments, diagnoses, and giving vaccines.

And then they would return to their home villages to serve the people they knew. They were still barefoot peasants like everyone else, but now with more skills.

The program was a wild success. By the mid-1970s, there were over 1.5 million barefoot doctors serving in China's villages.

Life expectancy over that period rose dramatically from 35 years to 63 years.

The developer version isn't going to be organized by a central government.

It only exists in very limited ways at the moment.

To explain this, we need to think about the scale of people interested in programming.

On one side, we have regular “end-users” of computers.

Now, in theory, I’m a big proponent of end-user programming, but I have to admit most of these people don’t give a shit about programming and don’t want to learn it. I think we need to stop harassing normal nurses, teachers, and therapists to code when they don’t want to.

So anyway, this is not about end-users.

It is also not about professional developers on the other side of the scale.

We all love writing code for its own sake. We will invent excuses to program.

Barefoot developers are going to be people who live in this middle bit.

At the moment, these are people like the teachers who make elaborate Notion spreadsheets for managing classes.

Or students who make over-the-top personal dashboards.

Or financial planning wonks producing extensive spreadsheets.

They are people who are technically savvy and interested in solving problems for themselves and people around them, but don't want to become fully-fledged programmers.

They still live within the world of end-user-facing applications.

At the moment, they rely on low and no-code tools. And they do wildly complex things within them, pushing these apps to their limits.

They are the kinds of people who would be thrilled to have more agency and power over computers.

But they never make it over what I call the command line wall.

They never end up in the terminal, because that is a huge jump in complexity, usability, and frustration from using something like Airtable or Notion.

This means most of their work is held hostage in the cloud and requires them to pay monthly subscription fees to access it.

They have far less agency and power over their creations than full-blown developers.

But I have this dream for barefoot developers that is like the barefoot doctor.

These people are deeply embedded in their communities, so they understand the needs and problems of the people around them.

So they are perfectly placed to solve local problems.

If given access to the right training and tools, they could provide the equivalent of basic healthcare, but instead, it's basic software care.

And they could become an unofficial, distributed, emergent public service.

They could build software solutions that no industrial software company would build—because there's not enough market value in doing it, and they don't understand the problem space well enough. This is the long tail of user needs.

And these people are the ones for whom our new language model capabilities get very interesting.

I'm interested in the question of what language models can do for barefoot developers.

This is a now infamous tweet by one of OpenAI's co-founders, Andrej Karpathy, claiming "The hottest new programming language is English."

What he meant by this is that these models can accept natural language input in the form of descriptions of interfaces or software functionality and output working code.

We are still in the very early days of this, but there are already a few prototypes out in the wild that show what's becoming possible.

This is Vercel's Vo . It generates working interfaces based on text descriptions of what you want.

I've told it to make me a personal finance dashboard with feature cards showing me my daily purchases, my recent transactions, and my income over the last couple of months.

It's given me three versions to pick from.

I can add follow-up instructions for how I want it to iterate on this interface. This gives me a good feedback loop.

I asked to add accounts and bills pages to the sidebar. It produced working code I can view and edit.

I can also directly select elements and ask it to edit them.

Another good example of this is TLDraw 's Make Real prototype.

I'm lucky enough to hang out with them sometimes and – unbiasedly – I think they're doing some of the most interesting work in generative interfaces.

The odd thing is they're nominally just building whiteboarding software.

But they hooked up a multi-modal language model (GPT-4 Omni) to their canvas so that you can draw any kind of interface you like and

annotate it with red drawings and text to tell the system how you want it to work

And then click this “make real” button and it will make your selection into a real working piece of software.

Here’s another one that instantly makes a photo booth app with a live camera feed from your device.

This required no custom code to make work. GPT-4 Omni just made it work.

Because this environment is an infinite whiteboard, you can select your current prototype, add annotations to it, and generate new versions.

This Make Real tool is online – you can go use it now.

If you're someone who's skeptical about the capabilities of language models, I suggest you spend some time playing with this.

This all feels exceptionally promising for the future of local, home-cooked software.

Remember that the stuff barefoot developers need to build is not as complex as professional industrial software we're all used to working on.

They don't need to scale up to millions of people, juggle conflicting user needs, or pivot their business plans, or ship lots of features very fast to make a high return for venture capitalists.

Most of the software needs of local communities could be solved with simple CRUD apps persisting data over time, with some basic user authentication, and a few API calls.

So as much as I expect the number of professional developers to grow in the coming years.

I expect the number of barefoot developers to grow exponentially more.

But to get to that world of millions of barefoot developers, we have some problems to work through in the current moment.

And the way I'm going to put this is that language model legos need glue.

Language models give you a bunch of disconnected lego pieces.

It can make pretty interface elements for you, it can manage state for you, make API calls, and write basic logic. But it doesn't tell you how to

stick all these things together into a working application.

Meanwhile, you want a cool castle, and it's very unclear how you are supposed to make a castle from these disconnected lego pieces.

At the moment, you still need the knowledge of professional developers to glue them together.

If you generate something in Vercel's Vo or TLDraw you still have no idea how you might deploy that to a particular web domain or to your iPhone.

Or persist data by setting up a database. You probably don't know what a database is.

Add multiplayer collaboration

Have multiple users log in with different view and editing permissions

And 99 others I haven't thought of that Hacker News will tell me about later.

We're missing the glue that brings all these pieces together into working software. And the glue comes in two forms.

We first need language model agents that are designed to act as central orchestrators for home-cooked software projects.

These agents can guide barefoot developers through the process of writing technical specifications and help them work out what kinds of tools they might need for a piece of software.

And then we also need tools that are designed to talk to and work with these orchestration agents.

I expect these agents will have a set of default tools they've been taught how to use and call on whenever their human needs to build some new software.

These might be APIs to multiplayer collaboration infrastructure, database managers, and deployment pipelines.

Which brings me back to the topic near and dear to all of your hearts. Local-first.

If I'm at all right about there being a sudden explosion of local, home-cooked software, how does this affect the strategy of the local-first movement?

How would you try and make sure local-first is baked into this future by default?

Because this default tool set for barefoot developers should really have a database that's a local-first database, right?

And once their apps are deployed, it would be nice if they worked offline, right?

Barefoot developers might not know to ask for these things in their technical specs or prompts. Whatever defaults are baked into these agents and their available tools are going to make a lot of decisions for them.

If you're currently building local-first tools, you should consider what kinds of interfaces would make them accessible to barefoot developers and their future agents.

Can someone prompt their way into a local-first setup in plain English? Could they write a technical spec in markdown and end up with a working database?

Could you develop your own small language model that guides someone through setting up your tool?

Frankly, can you stick TLDraw onto the front of your system and start letting people build full-stack apps with it?

To give you some motivation, I want you to imagine a future where we do get the explosion of local, home-cooked software, but it's not local-first.

We get powerful, flexible tools that let barefoot developers make lots and lots of local, home-cooked software. They solve all kinds of specific, local problems for the people around them. Their communities love it. They become dependent on what they've built.

But the software and the data behind it are all held in the cloud. And you have to keep paying a monthly subscription fee to access it. And then suddenly the terms of service change. And there's a giant advertisement in the middle of every homepage. And the subscription fee doubles.

And it turns out what they've built was never actually theirs all along.

I expect that would be a super lucrative business model.

So I think you should care about this because the local-first movement and the local, home-cooked software vision are distinct but philosophically aligned.

They're built on the same foundational values: that users should have agency and ownership over their data and software.

At the moment this community is focused on solving hard technical problems, but you should keep an eye on what's developing around you as parts of the software-making process rapidly become more accessible and democratized.

We can weave the local and local-first philosophies together and build tools that serve both of them. I really want local-first to be the default choice for local software. The alternative feels unquestionably worse.

The extent to which these two visions overlap and complement each other is up to everyone in this community to decide.

I want to end this with a quote by Ivan Illich, who I'm sure many of you have heard of.

He wrote a wonderful book called " Tools for Conviviality " where he talked about the importance of people being able to make tools for themselves.

He says, "People need not only to obtain things; they need above all the freedom to make things among which they can live, to give shape to them according to their own tastes, and to put them to use in caring for and about others."

Software is no exception to this.

That is the end of my talk. Thank you very much for listening.

I want to say this was a very challenging talk to write and it's the first time I've done it. I had to try to make some very big claims and back them up.

I would really love constructive criticism about places you think I'm wrong or I've overlooked important details. Just please be kind about it.

Mentions around the web

12 Likes and Retweets

Juha-Matti Santala mentioned **in Home-cooked, situated software**

July 20, 2024

For July 2024, James is hosting the IndieWeb Carnival and he picked the theme of Tools for us to write up: Over the next month, I invite you to write a blog post about tools and how they do, or have, influenced your createive process. You can write about tools in

obsidian_ns mentioned **in Incoming WE 2024-07-13** July 13, 2024

Links and posts [[2024-07-07]] Maggie Appleton paints a vivid picture of the possibilities that LLM's open up for non-professional developers to create software tools for their own problems in Home Cooked Software and Barefoot Developers. This resonates strongly with my hope for

Max Glenister mentioned

June 24, 2024

https://

Justin Sherrill mentioned **in Lazy Reading for 2024/06/16**

June 16, 2024

Text tools this week. CommonMark, a Markdown specification. LyX, a TeX editor. (this and previous from this thread.) nvi command summaries. Follow the thread. BSD User Group Düsseldorf Juli 2024. Programming Prayer: The Woven Book of Hours (1886–87). XScreenSaver 6.09 out now.

Want to stay up to date?

Subscribe via RSS Feed

The Garden
About
Now
Podcasts
Smidgeons
Library
Essays
Notes
Patterns
Talks
Colophon