The Dry Abstraction

Unitary Labs, Inc.

Abstract

The world runs on software, but software is very difficult and expensive to build. This leads to much dysfunction and impedes all kinds of progress. AI code generation technology helps, but not nearly enough. In order to address these problems, we have developed a new way to conceive of software called the Dry Abstraction. It lets people build software several orders of magnitude faster than conventional methods. This will eventually let people replace their current cacophony of services and internet platforms with a much simpler, unified platform that is hyper-customized to their own needs. This will unlock great amounts of innovation and accelerate progress along many dimensions.

Contents

Software is difficult to build. This is the root of many problems.

Producing seemingly simple software[1] often requires surprisingly large amounts of time and effort. Even simple software that one can describe and learn in minutes often takes a team of highly trained engineers months or years to build. Hidden from the user of most software is an array of technologies that must be integrated to create even a simple software service. These can include user interfaces, a server, multiple databases, a text search engine, user authentication services, and much more. Scaling and securing these services requires yet another handful of technologies to be integrated. Each of these technologies often requires nontrivial training and experience to be properly employed. Further, creating the user interface to these services often requires supporting many device types (e.g., phones, tablets, and laptops). The proliferation and adoption of new device types (such as smart watches, voice-based personal assistants, virtual/augmented reality, and car computers) along with demand for natural language-driven chat-based interfaces further exacerbate the problem.

The complexity of developing software has several negative consequences.

Obstacles to progress

There are many important projects that never happen because they require software development resources that people cannot find or afford. The result is that many scientific and technological advances do not happen, customer needs go unmet, industrial workflows remain suboptimal, and many other inefficiencies and missed opportunities occur throughout society[2].

---

[1] "Software" throughout this whitepaper refers broadly to software applications, websites, cloud software services, blogging platforms, social networks, search engines, crowdsourcing platforms, and various other artifacts created through computer programming.

[2] Companies eliminating large numbers of software engineers is not evidence that there is a surplus of programmers. It is rather evidence that they hired too many engineers who were not skilled enough to be productive for them. There is a surplus of engineers, but a shortage of sufficiently skilled engineers. The evidence for this includes high salaries for skilled engineers and a vast number of important software projects that never happen because they lack engineering resources.

There is an analogy between software production costs and energy costs. As energy prices rise, much growth and progress (in science, technology, business, and across society) slow. When the price of energy rises, the cost of producing and delivering goods increases, making them less profitable or unprofitable entirely. Thus, fewer people invest in these goods, and the economy slows. The price of oil (as of this writing) is approximately $70 per barrel. If the price of a barrel were instead $70,000 (and the price of other forms of energy correspondingly increased), the economy would implode, and a great deal of progress would halt. The cost of producing software is currently the equivalent of at least $70,000 per barrel. We don't see all the progress that is not happening because of that, because we have never seen $70-per-barrel software.

Monolithic, poorly customized, inefficient software

Because people cannot afford to build software optimized to their needs, they often resort to using a collection of monolithic software services that are not well-optimized for their specific requirements. Tasks that can take seconds or minutes with the right software often take hours or are impossible. To serve a large market, software platforms often include dozens or hundreds of features that a specific user will never need. This makes software difficult and cumbersome to learn.

Incentives to risk privacy

While producing software is expensive, people often expect it to be cheap or free. In order to recoup the cost of developing software, companies often resort to business models that rely on selling advertisements inserted into their products and/or exploiting user data. This generates incentives that often work against the best interests of the person using the software.

Advertising in software often puts the needs of the user in conflict with the needs of the advertiser. In other words, the software is often optimized for the advertiser as much as it is for the user. It is common practice for companies to limit the ability of third-party developers to build interfaces to their services in order to monopolize

the interface (and associated ad impressions). It is also common to make it difficult for users to migrate their data to another service. This leads to less competition and thus less innovation in the user interfaces for these services.

Siloed, unintegrated applications and services

Another consequence of monopolizing user data is that one must often use and awkwardly shift among several separate software services for a single task. For example, a typical small software startup will often subscribe to separate platforms for email, document collaboration, issue tracking, mailing list management, surveys, customer-relationship management, accounting, and more. Parents with children in schools must often interact with separate platforms for attendance, grades reports, parent-teacher conferences, homework tracking, camp signups, and more. Many such examples occur everywhere.

It is difficult for us to appreciate how much better this could all be. Before mechanized transportation and electronic communication, people did not reflect on how much inefficiency they had to deal with and how much progress they were missing out on because nonlocal communication would take days or months. Today we are like those people three hundred years ago. We have no idea how much our disintegrated and dysfunctional software situation makes us miss out on because we've never seen a world with simple, unified, hypercustomized, and functional software.

Malfunctions and security vulnerabilities

Software projects often require thousands of pages of computer code and a team of several engineers to coordinate their efforts. This leads to ample opportunities for mistakes that lead to oversights and mistakes in the product. These often result in malfunctions, some of which are security vulnerabilities. AI code generation has the potential to exacerbate this problem because so much code will not be understood by the people commanding the AIs to generate it. It will be even more difficult for them to anticipate and identify security vulnerabilities.

Lack of a competitive market for software and less innovation

It is common practice for software service providers to make it extremely difficult for third parties to make interfaces for their users' data. This is often accomplished through APIs with limits on functionality and/or usage. The result is that a single company often has a monopoly for each category of data a person or organization maintains (such as expenses, group messages, and customer databases). This results in less competition and innovation in software interfaces for that data.

Further exacerbating this problem is that the high fixed costs of creating software are a large barrier to entry for competitors and are one major factor that drives the software industry (and much of the rest of the economy) towards monopoly. This additionally reduces the amount of competition and innovation.

Even in competitive software markets today, each competitor must work slowly because of how slow and expensive it is to create and update software. Even a seemingly simple change like switching from unary "likes" to enumerated "reactions" took Facebook months to achieve. This greatly reduces experimentation and thus innovation.

Current approaches are inadequate.

There are several approaches to addressing the problems caused by how complex and expensive it is to build software. They are helpful in some ways, but they are not nearly enough.

Artificial Intelligence (AI). When ChatGPT and AI coding assistants first appeared, many people concluded that AI would soon replace most software engineers and lead to an acceleration of new and useful software releases. However, while these technologies have helped some programmers be more efficient, one still requires teams of programmers to build most software. Code generation technology can be

remarkably helpful to engineers, but it does not reduce the price of creating hypercustomized, powerful software platforms by orders of magnitude.

AI chat agent maximalism. Replacing all software with AI chat agents is not a total solution. The ability to use natural language chat with software can make many complex tasks much simpler. This will be very valuable and important. However, there will always be a need for graphical user interfaces. It is more efficient and effective, for example, to look at the time on a clock or inspect a route on a map or visualize data trends on a chart rather than to ask an AI assistant “What time is it”, “What is a good route to Springfield”, or “How does X correspond to Y”.

Training more software engineers. There is a limit to how many adequately skilled engineers humanity can produce, and we are reaching that limit. It is very cheap to learn remedial coding skills, but becoming an engineer skilled enough to drive and contribute to the most important software projects takes quite a bit of aptitude. Therefore, effective software engineering teams in organizations that create software are still very expensive and cost orders of magnitude more money than is available for most projects.

Open source software. Open-source software makes many programming projects much faster and cheaper than they would be otherwise. However, learning to use and customize open-source software for one’s project still requires a lot of time from skilled engineers.

No-code software platforms. There are many “no-code” platforms that enable people to create software without any coding. These do obviate the need for programming in some cases, but there remains an extremely broad array of software needs that these tools do not address. Fundamentally, this is because they are based on conceptual abstractions that are not powerful and flexible enough to match the broad variety of needs within software development.

The need for a new software abstraction

Abstractions make it much simpler to think about systems and faster to build them. For example, while all software runs on computers based on integrated circuits,

engineers virtually never need to think about what these circuits are when they are building software. Many probably aren't even aware that these circuits exist. They think in terms of programming languages, which are an abstraction of assembler languages, which are abstractions of machine code, which is an abstraction over integrated circuits. These abstractions make building software orders of magnitude faster than it would be without them. However, we can do even orders of magnitude better than that with additional abstractions. These abstractions must 1: reflect vertical integration and 2: be more aligned with how humans conceptualize software and interact with artificial intelligence.

Vertical integration

Many software services are built using a "software stack" that includes several components. Some typical examples are a database to store the service's data, a system for authenticating users, an application server for executing the service's logic, a web server for connecting the application server to the user interface, and that user interface.

Most efforts to make programming faster and easier optimize only one element of the software stack. For example, relational databases like PostgreSQL abstract over the database layer. Node.js does the same for web servers, as do React and Bootstrap for user interfaces.

This software stack is vertically disintegrated. Each layer is an abstraction that is not integrated with the other layers, and each layer of the software stack is relatively agnostic to the other layers. One can use Node.js to work with databases other than PostgreSQL, for example, and user interface systems other than React and Bootstrap.

While these vertically disintegrated software stack abstractions have accelerated software development to a degree, they are also an obstacle to further acceleration. The key reason is that many of the features of software people desire involve multiple parts of the software stack.

Two examples illustrate this point. First, consider “sharing”. One cannot simply find a “sharing” component to incorporate into their software project. This is because sharing requires multiple elements of the software stack that are rarely integrated. For example, sharing involves authentication (in order to confirm the user of the service is who they say they are), databases (to store the objects being shared and to store the access level of each user to those objects), a server (to decide which items to show to the user based on their permissions), user interface (to allow people to indicate what they want to share, and to whom they want to share it), etc. One cannot simply create a “sharing” software component that many projects can use because most projects have used different components at every element of the tech stack.

A second example illustrates that this need for vertical integration exists even for many of the smallest features of software services. Consider autosuggest (e.g., suggesting people to share a document with after the user has typed only the first few characters of a person’s name). To the lay person, this is such a common and intuitive feature that it’s not clear it requires much skill or time to program it. In fact, it does. This is because, like sharing, autosuggest involves orchestrating many parts of the tech stack. It obviously involves the user interface, but it involves the database (where the objects that are suggested are retrieved from), and permissions (to make sure that the user has permission to the objects suggested).

The vertically disintegrated software stack is just one example of how abstractions used in creating software are misaligned with how people actually think of software.

Aligning abstractions with how humans understand software

The fundamental reason that software is so slow to build is that computer languages are based on conceptual abstractions that use quite different concepts than people use to think about software. One way to characterize the job of a software engineer is to view them as translating a description of software in terms

that average people understand into computer code, i.e., a description of software that a computer can understand.

For example, a person might describe a software service like this:

“You have posts, posts have text, pictures, a user, etc. Each user has a set of followers. Anyone can follow anyone else, though people can block followers. There is a stream of posts ordered by recency. It only shows you posts by people you follow.”

It is the job of the software developer to translate this into (at the very minimum) the tens of thousands of lines of code it takes to implement this, including the database tables, queries, buttons, forms, routing handlers, password hashes, etc.

Ideally, we’d like to automate that translation. A person would merely need to utter a description in human language, and an AI would write the code and configure the hardware to implement this software. This would make it several orders of magnitude faster for people to create software. As mentioned above, AI has made serious progress in this direction for relatively simple and often-repeated programming tasks, but it is not yet close to adequate for the vast array of important software projects.

The Dry Abstraction

To address all these problems, we have developed a new framework for thinking about software called the Dry Abstraction. It makes it several orders of magnitude faster and easier to build software.

The abstraction applies to a very broad and important set of software. It includes much of “Software as a Service” (SaaS) software as well as social, search, and crowdsourcing internet platforms. We call this common software[3].

[3] People often dismissively refer to such software as “CRUD”. The acronym refers to the actions create, read, update, and destroy. The implication is that creating this software is relatively trivial. In fact they are wrong about this along several dimensions. There remains an extremely broad demand for this type of software and it is always much harder to implement than anyone anticipates. Apparently simple CRUD software often take months or years of teams of highly trained engineers to make useful. Such software actually requires a large array of other

There are three insights that motivate the Dry Abstraction.

1. Common software can be thought of as managing a database of items that fit an ontology (i.e., they belong to types, have standard fields, and can be organized into categories).
2. There are many common operations that are performed on this database across common software.
3. One can automatically generate a user interface for this database based on its ontology.
4. The differences among common software is mostly characterized along four dimensions: ontology, interface, permissions, and workflow.

Our approach is to provide a platform that:

1. Is a database that performs common operations.
2. Automatically generates a (graphical and chat) interface for the database.
3. Makes it very easy for people to customize the ontology, user interface, permissions, and workflow of the database.

Such a platform lets people build and customize the software services they need orders of magnitude more quickly than they would using conventional software methods.

Common software manages a database that adheres to an ontology.

Common software can be thought of as managing a database of items. For example, email clients manage a database of emails. Issue trackers manage a database of issues in a software project. Social media platforms manage a database of connections among people, along with the posts they make.

---

operations (such as sharing, authentication, searching–and more recently–chatting, recommending, etc.) For this and other reasons, there is an extremely broad need for CRUD software that is unmet, even in organizations one would expect to be able to afford to hire engineers to build them.

Each kind of service has an ontology composed of types, fields, and categories.

Software services manage several types of items. For example, some item types in a social media platform are profiles, connections (such as following or friendship) between profiles, posts, comments on posts, and reactions to posts. In a task manager, two item types are tasks and the people they are assigned to.

Each item type has a set of fields that can characterize it. For example, fields in an email are the sender, the receiver, the subject, the body, etc. Fields in a task in a project manager are the description of the task, the due date, the person to whom they are assigned, etc.

Items are organized into categories. For example, email folders are sets of emails, group messaging channels are categories of messages, and folders in file systems are sets of files. Categories are often organized into subcategories. For example, the folder/subfolder relationship is a kind of category/subcategory relationship. Categories are often a locus of data access permissions. For example, the members of a channel in a group chat platform are typically the only ones who can view and add messages there. In cloud drive platforms, one often gives people access to large numbers of files by putting those files into a single folder and sharing access to that folder.

A less obvious point is that items themselves often serve as categories or containers for other items. We call this object-category duality (OCD). For example, in a review platform (e.g., for business or films), one can think of the reviews as being contained in the object (item) they are reviewing. Likewise, in platforms where people comment on items (e.g., a social media post or an issue in a bug tracker), the comments can be thought of as being contained by the item they're commenting on.

Common operations

You can think of what software does in terms of operations on the items in the database that the software manages. This is an underlying insight of object-oriented programming. The insight in the Dry Abstraction is that there is a

single set of common operations that characterizes a vast array of software use cases. Some of these operations are: View, Edit, Delete, Filter, Search, Interrogate, Share, Categorize, Notify, With, Rank, and Recommend items.

The default, ontology-driven interface

An important part of a software service is the interface people use to perform common operations. While there are many superficial interface differences among software services, there are many substantial similarities. Because of this, it is possible to programmatically generate a fairly powerful and functional user interface for a database from its ontology. This database can be generated without any additional coding. Thus, when one specifies an ontology for a database, they are also specifying a full-stack software service with a user interface that manages the items in that database.

Here are some examples of how particular ontological elements have standard manifestations in user interfaces. Many software services include a sidebar that has categories and a main section that lists a stream of objects. Editing items involves filling forms, and the elements of these forms are fairly predictable given the type of data the form element corresponds to. For example, dates are often entered with a calendar pop-up, colors with a color picker, strings with a text box, etc. The permission rules governing the service also predictably affect these elements. For example, one is not given an input element for fields that they do not have access to.

This is also true in natural language chat. Many common ontological elements have common verbal manifestations. For example, the ways of speaking about addresses and dates are quite similar across domains. The address of a warehouse in an inventory management chat interface will be interpreted in the same words and grammar as an address used in a contact management interface. Likewise for dates: e.g,. shipping dates in inventory software and for birthdays in a contact database.

Four fundamental kinds of differences

What makes common software services different from each other is how they implement the common operations. While there are many superficial differences among software services, there are four particularly important kinds of differences. These pertain to the ontology, permission rules for access to items, workflow, and interface customization. Our thesis is that most of the essential differences among common software services fall into these categories. If this is true, we can describe a software service just by mentioning these differences and simply presupposing the common operations.

Ontology. One way software services differ from each other is the kinds of items they contain and the fields these items have. For example, emails have senders, receivers, a body of text, etc. Social network posts often have pictures, links, text, etc., Contacts in contact managers have names, addresses, emails, etc.

Permissions. Software services generally implement policies about people's access to items that are different from other services. For example, in email platforms, only the sender can edit a message, and once it has been sent, it cannot be edited. Also, only the sender and receiver can view the email. In many group messaging platforms, anyone who is a member of a channel can see all the messages in it, but only the author of the message can edit it. Unlike email, they can edit it after it has been posted. In some social networks, the only people who can read a post are the person who authored it and their friends. In others, anyone who follows a person can see their posts. In issue trackers, it is common for anyone on the team to view an issue but only for a manager and/or the person assigned to it to be able to change its status.

Interface customizations. Although interfaces in software services have many fundamental similarities, there are, of course, differences. For example, the list of items in a folder containing emails (e.g., the inbox) in a mail client is often tabular, while in social media services, posts are often rendered with "cards" that prominently feature pictures from a post. Ontologically similar elements are often rendered differently in interfaces for different tasks. Consider booleans, for

example. In task managers, user interface cards representing a task often include a checkbox to easily mark a task item as complete. In other services, a boolean is represented with a toggle.

Workflow. Software services differ according to when the common operations are performed. For example, in a hiring/recruiting service, a job application might be sent to a hiring manager who then has the option to send it to a few possible reviewers. When an interview is scheduled, a notification to the interviewers is sent. Most or all of what happens in these workflows can be characterized in terms of the common operations; what is different among services is when they are performed.

Our thesis is that there is a small set of such differences that characterize the essential differences among common software services. The above list of four kinds of such differences is our best current guess at this set, though it might change as we gain more experience with the abstraction.

How broad is the abstraction?

The Dry abstraction characterizes a very broad array of software. We have used it to build working prototypes of many of the following types of services and have sketched out most of the rest.

- Personal. Fitness/diet/health tracking, recipe database, note taking, car and home maintenance tracking.
- Work collaboration. Project management, KPI tracking, bug tracking, scrum boards, recruiting dashboards, mailing list management, surveys, expense tracking, document management.
- Businesses. Customer support assistants, recruiting/interviewing/hiring pipelines, maintenance process management, inventory management, customer relationship management (CRM).
- Events. Dinner/party scheduling, conference organization, event directories, meetup directories and organization, potluck signups, wedding hubs.

- Friends and family. Photo sharing, trip planning, chore tracking, child activity scheduling.
- Marketplaces. Online stores, two-sided marketplaces (e.g., ride sharing, AirBnB, etc.), classifieds, job boards, online dating.
- Students and education. AI chat knowledge bases (e.g., NotebookLM, Projects on Chat GPT or Claude, etc.), dynamic syllabi, exam administration, citation generator, research hubs, parent-teacher communications, grade reporting.
- Crowdsourcing. Crowdfunding, crowdsourced vertical search engines, rating and review platforms (like Yelp, Rotten Tomatoes, etc.), scientific research data collection and dissemination.

Related metaphors

The following are alternative metaphors for conceiving of software that include the insights underlying the Dry abstraction.

ChatGPT/Claude 8.0. The first versions of AI chat platforms such as ChatGPT and Claude enable conversations based purely on the knowledge they acquired during their training. Then, these platforms and many others (such as NotebookLM) let you chat with a database of text items, such as documents and web pages. One can think of the Dry Abstraction as what would result from adding several major features in subsequent versions:

- Ontology. Rather than letting people interact with a database of text, allow them to interact with databases of items conforming to an arbitrary ontology that the user can specify.
- Customizable ontology-driven user interface. Expand the user interface currently used to manage text sources to manage and organize new types of items specified by users, and let them customize this user interface.

- Customizable, granular sharing and permissions. Let teams collaborate with the same database and allow end users to establish permission policies on the types of items in the database's ontology.
- Workflow. Allow people to specify when sequences of common operations should occur.

Semantic Dropbox. Cloud drives are essentially databases of files. They include a default interface. They are organized into categories (aka folders). One can perform many of the common operations on them (adding, deleting, searching, sharing, etc.) If they include an AI chat interface and let people specify and manage items of arbitrary ontological type, customize the UI, set permission rules, and establish workflows, they would have the main features of the Dry Abstraction and let people implement a wide variety of software services.

Dry.ai platform

Dry.ai is our attempt to embody the Dry Abstraction into a platform that lets people quickly create their own hyper-customized space of software services without any coding.

Dry.ai lets you set up "smartspaces", which are databases of items where you and others can collaborate. Items in a smartspace have types and fields and can be organized into folders. You can specify arbitrary item types with a type editor. You can use the type editor to specify how items are surfaced in the user interface. There is a permission table that lets you specify permission policies for each type of object. On the page for each folder, you can make "tabs" that let you specify which items are seen and how they are laid out. There is a chat interface for performing many of the common operations on the items and asking questions about them. User account management is built in. We have thus added a few minimal workflow capabilities, but they are relatively minimal.

By defining types, customizing them, adding folders and tabs, and establishing permission policies, it is possible to create powerful AI-enabled software services within minutes or hours in Dry.ai, without coding. Software platforms that would

take conventional software engineers (even those using AI coding assistants) weeks or months to build can be built in Dry.ai in hours with no coding. To make building even easier, Dry has a templating capability that lets people clone existing Dry smartspaces and quickly customize them to their own specific use cases. This pattern – clone and customize – makes building in Dry even quicker and easier.

Although there is quite a lot of work left to realize the full Dry vision in Dry.ai, people have already built a wide variety of powerful software services with it, and it is being used by real users ranging from individual hobbyists to large organizations.

Our experience to date gives us confidence that a more mature version of our platform will soon consolidate numerous fragmented, outdated services into a single, unified solution: one that's precisely tailored to meet individual and group needs and preferences.

The Dry abstraction enables dramatic improvements to software.

Today, people must adapt themselves to a disintegrated collection of software siloes; they do not have the time and resources to create a simple, uniform software platform customized to their specific needs. Software platforms built around the Dry abstraction change this. They let people build the software they need, as soon as they need it. As soon as their needs change, they can almost instantly change the software accordingly.

A single unified space

In terms of the Dry abstraction, each individual software silo people use today is a database of several types of items. It is difficult to integrate them. APIs and other methods of integration fall far short.

With a platform built around the Dry abstraction, you can easily create a single software platform that has the capabilities of each of these silos. This platform

would be a single database that includes all the types of items people need to deal with.

This makes software much simpler and more effective in several ways.

Unified organization. Folders in a cloud drive, task manager, email client, and issue tracker are all ways of organizing items in each platform. Synchronizing this organization between platforms is difficult and rarely happens. This makes it more difficult to accomplish many tasks. When all the items are in a single database, creating and maintaining information is much easier.

Unified search. Finding items that are conceptually related often requires someone to search over several software platforms. When all the items are in one place, the search becomes an order of magnitude faster and easier. You just need to search once.

More powerful AI assistants. When AI assistants have access to data in only one software silo, it is much more difficult to accomplish tasks using all the relevant data available across platforms that an individual or organization is subscribed to. When all data is structured and organized in a single database that AI assistants have easy access to, they can be much more powerful.

Streamlined interfaces and workflows. When there are no artificial limits or boundaries between collections of data, users can set up interfaces and workflows that are optimized to their specific situation and either totally automate many cumbersome tasks or make them more streamlined.

Hyper-customized, simpler

When you can create your own software easily as soon as you need it, it will also be much simpler to use and will more powerfully meet your specific needs.

Exactly the features you need. The software will have just the features you need, and they will work the way you need them to. You design it to optimally address your use case rather than have to adapt yourself to suit software someone else built

for a mass audience. This will make you considerably more effective at whatever you are aiming to achieve.

No excess or clutter. Much software today creates vastly more features than any specific person or organization needs. It must include these to serve a large market. When you can create your own software, it will include only the features and depth you need. That makes the software much simpler to use and easier to learn.

AI agents that make complex tasks simple. When people build their own software using the Dry abstraction, that software will include AI agents and be compatible with various LLMs. Thus, Dry can accelerate the adoption of AI agents that make complex tasks simple.

Change your software as soon as your needs change. When your needs change, you will be able to change the software immediately to adapt to those changes. Slow software development will no longer impede change and improvement.

Experimentation. It is often difficult to anticipate the best way for software to work in a specific situation. Today, that means if an engineer or product manager guesses wrong, everyone must deal with the consequences for a long time. It also means that people take too long to implement useful capabilities for fear of doing it wrong. When you can make and revise software almost instantaneously, you can experiment much more. This results in much better software.

More secure

When software does not take a large team of engineers months to produce, and when it doesn't involve millions of lines of computer code, there will be orders of magnitude fewer opportunities for errors and unintended security vulnerabilities. Also, when people can make their own software, they will be able to control their own data and no longer have to trade privacy for convenience.

More innovation

When anyone can build the software they need, quickly and cheaply, there will be more competition, ideas can be explored orders of magnitude more quickly, and thus there will be much more innovation.

The Future

Letting people create the software they need, as soon as they need it, will unlock a vast amount of human potential and dramatically accelerate progress. Dry.ai is our attempt to make this happen as soon as possible.