Proposal For a Modern Software Ecosystem

Peter Alexander, ProDataLab

March 13, 2017

Abstract

Utilizing state-of-the-art cryptocurrency technologies and an unconventional programming model, a complete paradigm shift in technology production is possible. Time-to-market for all software projects could decrease by a hundredfold. It is strongly believed that the innovations presented below, will change the way all computing is done in the future. This truly amazing efficiency would not be possible without the ability to stand on the shoulders of giants.. the open source software community.

Overview

The Problem

Software is very much infused in our society. It is the technology that enables all other technologies and is therefore what stems all increases in societal productivity. Soon, with the advent of technologies like artificial intelligence, data mining and the internet of things, that notion will be enhanced by magnitudes. Yet, even a less than an astute obverser can easily see that, the creation and usage of software is very problematic and should be vastly improved upon. The conventions in today's software usage, sales, production and execution are certainly frought with inefficiencies.

Creators of software often voice frustrations of unwarranted complexity and lack of interoperability. It is necessary to know what are essentially redundant programming systems, where their favorite system would or should suffice. Any supply side participant can easily attest to the replication of effort that is both profuse and pervasive in today's environment.

Users of software often voice frustrations with the time and sophistication required to learn a software system.. that the documentation or help system isn't of much help. All too often, a software system lacks a feature or is too featureful. Small business owners may find navigating the obstacle course of software system selection and integration impossible for mere mortals.

Other aspects of concern and inefficiency can be seen in the market place itself. In today's environment, these market places are consentrated silos and are oligopolistic. A few behemoth sized corporations take large percentage commissions from a lowly software producer's product sales. They dictate rules that are at times overly stringent or simply used to control the domination they hold in the market. Users may also find it intimidating and inconvenient to hire a software developer for a bespoke project.

We ask the reader to take a moment and think of their own experiences with software. Have you ever found yourself frustrated? Have you ever considered some aspect overly complicated, unintuitive or inflexible and thought to yourself "it shouldn't have to be this way"? The current software ecosystem greatly needs a paradigm shift and is ripe for innovation.

The Solution

An elaborate and concise proposal is put forth to vastly improve the current software paradigm through a world wide collaboration called CodeDepot. This paper begins a conversation towards the direction of the needed improvements and we ask the community to participate. This community that is mentioned, are the people that engage with software whether it be in its production, sales, tutelage or usage.. whether they are casual or power users. Having discussion and contribution globally and easily will ensure that the best solutions to problems we all identify, will be addressed most thoroughly and effectively.

The solution first entails the technology of an unconventional programming model, the leveraging of open source software and incorporation of the ground breaking technology underlying Bitcoin.. the blockchain. Second, through a network-private currency, an economic model incentivizes collaborative participation in software creation, usage, tutelage and governence. Anything that can be incentivizeded.. will be incentivized. Overall, and most importantly, it is designed to find efficacy and efficiency for every single aspect of a decentralized and ubiquitous software ecosystem.

Below are six foundational pillars of the proposed software ecosystem that may forever change the entire paradigm of our engagement with software. Again, what is presented at this time are merely proposals that will be refined by any willing individual participant and all the participants in whole. The paper and its addendums will also introduce other worthy aspects and notations pertinent to the execution of this proposal.

The six foundational pillars are comprised of:

1. **CodeChain**, A distributed computation engine that provides significant advancements.
2. **CodeDepot**, A decentralized market place for developers and end users.

3. **WorkSource**, A decentralized software market place for end users to hire freelance developers.
4. **CodeSweet**, A suite of developer tools designed to simplify the developer experience.
5. **CodeCoin**, A cryptocurrency token for incentivizing and monetizing goods and services.
6. **CodeDepot Collaborative**, A decentralized governence model for the CodeDepot ecosystem.

Open source software and its community model is one that is quite nobel. It is quickly becoming the defacto standard world wide even amongst corporate giants and governments alike. CodeDepot is wholy inspired by this model and will monetarily incentivize and reward the open source software community. It will also project open source software's ethics and goverence in a decentralized manner.

You are encouraged to envision the possibilities of a collaborative effort to change the current paradigm surrounding the production, consumption and usage of software. You are encouraged to join this effort to modernize the entire software ecosystem via open source software and its principles. You.. the creators and users of software are asked as to what you would have improved. It are these items that will be monetarily incentivized and addressed in decentralized collaborative called CodeDepot. Become a contributing member of the CodeDepot Community.

The Impact

Within five years time, CodeDepot will have sparked a paradigm shift throughout the entire software ecosystem.

- Time-to-market for software projects will have decreased a hundredfold.
- The complexity of developing software will have been mostly eliminated.
- Snippets of software are in essence Lego like reusable components that just snap together, even autonomously into a desired structure.
- It will not matter which programming language was used to create an individual component and any component will seamlessly communicate with others.
- Computation will efficiently execute at a climax performance.
- Once a component or a structure of components has been created, it will never have to be created again by anyone. Yes, ever again.. as in the literal sense of forever.
- An individual component of code will be shared by every piece of running software in the world that requires it.
- An executing software system, even a mission critical system, could be altered or replaced without even a nanosecond of downtime.
- User interfaces will be so easy to use.. a four year old can run the world.

- User interfaces will also be component based.. meaning flexibility, device agnostic, and fully customizable.
- Compensation for all the value added to the ecosystem will be decentralized with intermediaries removed.
- Propietary software will all be but eliminated and open source will be the convention.
- The most cutting edge technologies will be infused into software development and usage.. as in the various use cases for artifical intelligence and virtual reality to name a couple.
- Computing and data storage will be most secure with privacy protected.
- Conventions and standards will be decentralized and evolve autonomously.
- Governence of the ecosystem will be decentralized.
- Our overall engagement with software will finally be a seamless experience that is quite pleasant.

Living Systems

The Six Pillars

CodeDepot Collaborative

Dedicated To Pieter Hintgens

In his book "The Wisdom of Crowds", James Surowiecki wrote, "under the right circumstances, groups are remarkably intelligent, and are often smarter than the smartest people in them." He noted that a collective intelligence usually produces better outcomes than a small group of experts, even if members of the crowd do not know all the facts or if they individually choose to act irrationally. To put it another way, a group of random people will on average be smarter than a few experts. It's a counterintuitive thesis that mocks centuries of received wisdom. So together, an engaged society will undoubtedly find the best solutions to the issues that face them. Over the past forty years the open source software community has shown that open participation provides better solutions to that of a closed system. The longevity of decentralized living systems, which self organize and are able to adapt rapidly, are much greater than that of planned centralized systems.

The CodeDepot Collaborative is the working name of proposed open goverance model for the CodeDepot ecosystem; where directives and decisions are decentralized and democratized amongst the entire community. This notion of decentralized goverenance and transparency is aimed at stemming the problems of any centralized governing system that inherently creates hierarchical levels of domination and control.

The collaborative will utilize an infrastructure which allows for most easy access and communication. Any barriers that may cause friction of any sort in the goverance process will be sought after and removed collectively. Members that are needed in positions of trust will only earn those positions meritocratically and only through community approval. Every member shall have a voice and opportunity for rapid advancement.

Bitcoin's underlying technology of the blockchain, enables a radical departure from the governance model of the old world. It follows trends of decentralization that have emerged through the internet in recent years including Bittorrent, the open source movements and collaborative production platforms like Linux and Wikipedia. The essence of this game changing invention is distributed trust (no need for third party reconciliation). Blockchain solves the scaling issue of trust. With its decentralized security, we can now create a more open and inclusive society at a global scale.

Bitcoin's decentralized system opens a door to a new paradigm where people can choose to abide by a protocol of consensus which is a different from the logic of domination and control of a centralized paradigm. Decentralized Autonomous Organizations, or DAOs, a new concept now available through this cutting edge technology, will provide many benefits to open and decentralized organizational structures where they are applicable. But, technology can't solve everything. Technology is just a tool. It always needs to be accounted for by democratic consensus of people. Technology should never be used to replace human interaction and connection.. it should be used only to enhance it.

In short, a collaborative is similar to a cooperative which is a legal entity defined as "a jointly owned enterprise engaging in the production or distribution of goods or the supplying of services, operated by its members for their mutual benefit". And.. in this case, we include consumers into our definition of a collaborative as well. It is a very interesting question, but quite plausible at this time, whether the concept of a DAO is able to supplant any possible need for the collaborative to be a registered legal entity.

CodeCoin

The invention of blockchain technology in 2008, has provided for the world a whole paradigm shift in financial and contractual mechanisms. It is truly a marvel that will disrupt many incumbent institutions. Systems that once required intermediaries and centralized solutions can now be peer to peer and decentralized. Trust of persons and entities unknown can know be established without the need of 3rd parties for verification or validation. Moneys can now be sent digitally without any concern for fraud or corruption. Ability to make payments in fractions of a penny are now facilitated. These are truly amazing times and we as a community of software engagers can benefit from this remarkability.

At the core of CodeDepot is a network embedded cryptocurrency, enabled by blockchain technology, in which a well formulated economic model with a monetary policy is inherent. The tokens of the cryptocurrency are of limited supply, so as the network function provides more and more utilitarian value, the monetary value of the token will increase. The utilitarian potential of the CodeDepot proposal is most exciting. Thus, the token's value increases as the utility grows.

The economic model proposed is rich with monetary incentives for all contributors who increase adoption and utility. Anything that can be incentivized will be incentivized. Every snippet of code deposited and hence executed will be rewarded. Every supporting document and usage efficiency will be rewarded. All marketing that inceases adoption will be rewarded. The benefits of a builtin network currency with its own, well considered, monetary policy is an exiciting tool that feeds incentivization and therefore reward for all contributors.

CodeDepot will also be a marketplace for end-users to purchase software and related services using the tokens of the cryptocurrency. These end-users are inclusive all types of individuals, small businesses and enterprises. See CodeDepot section below.

CodeDepot

The core component of the ecosystem is a marketplace where users and producers engage. This is composed of two parts where developers will have an interface to deposit software code into the system's repository and users will have an interface to utilize it. Other participants will include documentation contributers, article writers, bloggers, audio and video producers. Actually, anything surrounding software and related technology.

As described in the section for CodeChain below, developers will be rights holders for what essentially are snippets of code called components in this paper's nomenclature. They will use CodeDepot as a repository and market interface for their components of code and other contributions.

Users will have many advantages over conventional software produced today including but not limited to:

- Flexibility
- Scalability
- Security
- Ease of use
  - Well written documentation
  - Support videos
  - interactive autonomous tuteldge for every feature
  - Friendly wizards for configuration needs
- Accessibility
- Privacy
- Complete control and ownership of their personal data.
- Mechanisms will be built-in that provide the end-user and easy ability to:

    - Give seamless, instantaneous feedback.
    - Directly request alterations or needed features.
- Intuit the product through complete documentation and autonomous tutelage.
- Hire affordable freelancers directly from their user interface.
    - For assistance or instruction.
    - For software developers to provide any possibly needed customization.
- Possibly choose "payment methods"
    - Advertising
    - Onetime fee for apps
    - Per execution cycle (micropayments)
    - Contractual, eg. monthly/yearly
    - Synergies via CodeDepot's economic partners
    - Selling value of their usage characteristics e.g., Facebook business model, where the user is the product

CodeChain

> *"A new scientific truth does not triumph by convincing its opponents and making them see the light, but rather because its opponents eventually die, and a new generation grows up that is familiar with it."*
> ~Max Plank's Principal

The problems with conventional programming paradigms are numerous. In fact, at the time of the initial draft of this paper, querying Google's search engine for "problem AND programming AND language" produced 80.5 million results[1]. With many articles titled similar to "The Problem with Programming"[2], It seems fruitless to itemize these problems as a comparison to what is proposed here, let alone the problems of the entire software centric paradigm. Instead, as an introduction, the following questions are asked of the reader:

1. What if snippets of software were in essence Lego like reusable components that just snapped together, even autonomously into a desired structure?
2. What if once a component or a structure of components was created, it would never have to be created again by anyone? Yes, ever again as in the literal sense of forever.
3. What if once a component was created it would be shared by every piece of running software in the world that required it?
4. What if an executing software system, even a mission critical system, could be altered or replaced without even a nanosecond of downtime.
5. What if it did not matter which programming language was used to create an individual component and that any component could seamlessly communicate with others?

[1] [@__problem__0]
[2] [@pontin__problem__0]

6. What if a software producer's time-to-market was reduced by a hundred-fold?
7. What if a software's execution was most reliable and the most possibly secure from intrusion?
8. What if a software user's privacy was held in the highest regard.

CodeChain, it will be shown, is a system that could and will provide these desirable properties, as well as others.

The Architectural Inspiration

> *"Forty-five years have now passed since J. Paul Morrison*[3] *first discovered Flow-based Programming, and the programmers who worked during the beginning of the Von Neumann era are dying off. In their place, a new generation of programmers frustrated with the shortcomings of the traditional paradigms is coming of age, and theyâ€™re willing to try something new."*[4]

The rest of this section is quoted from the Wikipedia entry on Flow-based Programming[5]

In computer programming, flow-based programming (FBP) is a programming paradigm that defines applications as networks of "black box" processes, which exchange data across predefined connections by message passing, where the connections are specified externally to the processes. These black box processes can be reconnected endlessly to form different applications without having to be changed internally. FBP is thus naturally component-oriented.

FBP is a particular form of dataflow programming based on bounded buffers, information packets with defined lifetimes, named ports, and separate definition of connections.

Flow-based programming defines applications using the metaphor of a "data factory". It views an application not as a single, sequential process, which starts at a point in time, and then does one thing at a time until it is finished, but as a network of asynchronous processes communicating by means of streams of structured data chunks, called "information packets" (IPs). In this view, the focus is on the application data and the transformations applied to it to produce the desired outputs. The network is defined externally to the processes, as a list of connections which is interpreted by a piece of software, usually called the "scheduler".

The processes communicate by means of fixed-capacity connections. A connection is attached to a process by means of a port, which has a name agreed upon between the process code and the network definition. More than one process

[3] [@_john_0]

[4] [@_how_2013]

[5] [@_flow-based_2017]

can execute the same piece of code. At any point in time, a given IP can only be "owned" by a single process, or be in transit between two processes. Ports may either be simple, or array-type, as used e.g. for the input port of the Collate component described below. It is the combination of ports with asynchronous processes that allows many long-running primitive functions of data processing, such as Sort, Merge, Summarize, etc., to be supported in the form of software black boxes.

Because FBP processes can continue executing as long they have data to work on and somewhere to put their output, FBP applications generally run in less elapsed time than conventional programs, and make optimal use of all the processors on a machine, with no special programming required to achieve this.[6]

The network definition is usually diagrammatic, and is converted into a connection list in some lower-level language or notation. FBP is often a visual programming language[7] at this level. More complex network definitions have a hierarchical structure, being built up from subnets with "sticky" connections. Many other flow based languages/runtimes are built around more traditional programming languages, the most notable example is RaftLib[8] which uses C++ iostream-like operators to specify the flow graph.

FBP has much in common with the Linda[9] programming language in that it is, in Gelernter and Carriero's terminology, a "coordination language":[10] it is essentially language-independent. Indeed, given a scheduler written in a sufficiently low-level language, components written in different languages can be linked together in a single network. FBP thus lends itself to the concept of domain-specific languages[11] or "mini-languages".

FBP exhibits "data coupling", described in the article on coupling[12] as the loosest type of coupling between components. The concept of loose coupling[13] is in turn related to that of service-oriented architectures[14] (also see microservices[15]), and FBP fits a number of the criteria for such an architecture, albeit at a more fine-grained level than most examples of this architecture.

FBP promotes high-level, functional style of specifications that simplify reasoning about system behavior. An example of this is the distributed data flow model[16] for constructively specifying and analyzing the semantics of distributed multi-party protocols.

[6] [@_flow-based_0]
[7] [@_visual_2017]
[8] [@_raftlib_2017]
[9] [@_linda_2017]
[10] [@gelernter_coordination_1992]
[11] [@_domain-specific_2017]
[12] [@_coupling_2017]
[13] [@_loose_2017]
[14] [@_service-oriented_2017]
[15] [@_microservices_2017]
[16] [@_distributed_2013]

Flow-Based Programming was invented by J. Paul Morrison in the early 1970s, and initially implemented in software for a Canadian bank.[17] It went live in 1975, and, as of 2013, has been in continuous production use, running daily [without any interruption], for almost 40 years. FBP at its inception was strongly influenced by some IBM simulation languages of the period, in particular GPSS[18], but its roots go all the way back to Conway's[19] seminal paper on what he called coroutines[20].

Limitations of FBP

At a superficial level, FBP is an ideal programming paradigm that offers quite a few benefits over conventional paradigms. At scale though, there is a limiting condition of context switching[21], especially so on conventional general purpose central processing units[22] (CPU). An FBP paradigm at scale, a point will be reached where the number of context switches on a single machine, multi-core[23] CPU, overwhelms the system and causes notable latency[24]. On average, context switching costs approximately 30 microseconds of overhead per occurrence. One benchmark of the theoretical limitations of context switching has an upper bound of 18.75% of CPU cycles wasted due to context switching. Generally, optimal CPU use is to have the same number of worker threads as there are hardware threads when a process is CPU bound, whereas I/O bound processes permit more[25]. These considerations puts the FBP paradigm at very much a disadvantaged ideal of maximal efficiency.

Component Based Programming

In order to overcome the conditional limitations of context switching per processing node, in a strictly FBP paradigm, provided here are areas of consideration to help maximize the efficacy of the CodeChain system. The term Component-based Programming (CBP) is coined here for the purpose of a enlisting a stronger emphasis on components over that of data flow as it is for FBP.

Utilizing Software Translation

[17] [@_how_2013]
[18] [@_gpss_2016]
[19] [@_melvin_2016]
[20] [@_coroutine_2017]
[21] [@_context_2017]
[22] [@_central_2017]
[23] [@_multi-core_2017]
[24] [@_latency_2017]
[25] [@sigoure_how_0]

The concepts fundamental to FBP (autonomous blackbox components loosely coupled via lazy linkage) can be easily considered at the various phases of the compilation stack[26] prior to execution. Essentially, what this means is that we can remove the constraints from that of each component needing to be its own execution process[27] or thread[28], yet still be most loosely coupled. We can redefine networked inter-process[29] components to that of a virtual model[30], that can then be implemented by encompassing one or all of the compilation's translation[31] stages prior to execution.

1. Source code[32]
2. Lexical analyzer[33]
3. Parser[34]
4. Semantic analyzer[35]
5. Intermediate Representation[36] (IR) code and its linkage[37]
6. Optimizer[38]
7. Machine code[39] and its linkage[40]
8. Just-In-Time compilation[41] or interpreter[42] engine

General Purpose Graphical Processing Units (GPGPU)

Another most exciting and promising consideration, is to apply the notion of CBP to include that of specialty hardware processors like that of the GPGPU[43]. GPGPUs provide a processing model of thousands of concurrently executing threads. Utilizing these high-scale concurrent processors, one can imagine the promise of the original FBP concept of inter-communicating processes/threads[44] and lazy linkage, without the burdens of scalability that are imposed when merely targeting that of a CPU architecture. GPGPUs have very large register files which allows them to reduce context-switching latency. Due to this, register file size is increasing over different GPGPU generations, e.g., the total register file size on Maxwell (GM200) and Pascal GPUs are 6144KB and 14336KB,

[26] [@_compiler_2017]
[27] [@_process_2016]
[28] [@_thread_2017]
[29] [@_inter-process_2017]
[30] [@_virtualization_2017]
[31] [@_translator_2017]
[32] [@_source_2017]
[33] [@_lexical_2017]
[34] [@_parsing_2017]
[35] [@_semantic_2016]
[36] [@_intermediate_2016]
[37] [@_linker_2017]
[38] [@_optimizing_2017]
[39] [@_machine_2017]
[40] [@_linker_2017]
[41] [@_just_in_time_2017]
[42] [@_interpreter_2017]
[43] [@_general-purpose_2017]
[44] [@_inter-process_2017]

respectively. By comparison, the size of register file on CPUs is small, typically tens or hundreds of KBs.

GPGPUs are stream processors[45] â€“ processors that can operate in parallel by running one kernel[46] on many records in a stream[47] at once. Stream processing is a computer programming paradigm, equivalent to dataflow programming and therefore lends itself (almost) naturally to the concept of CBP. Such applications can use multiple computational units, such as the floating point unit[48] on a GPGPU without explicitly managing allocation, synchronization, or communication among those units.

The stream processing paradigm simplifies parallel[49] software and hardware by restricting the parallel computation that can be performed. Given a sequence of data (a stream), a series of operations (kernel functions) is applied to each data element in the stream. Uniform streaming, where one kernel function is applied to all elements in the stream, is typical. Kernel functions are usually pipelined[50], and local on-chip memory (cache[51])is reused to minimize external memory bandwidth. Since the kernel and stream abstractions[52] expose data dependencies[53], compiler tools can fully automate and optimize on-chip management tasks. Stream processing hardware can use scoreboarding[54], for example, to initiate a direct memory access[55] (DMA) when dependencies become known. The elimination of manual DMA management reduces software complexity, and the elimination of hardware caches reduces the amount of the area not dedicated to computational units such as arithmetic logic units[56].

Everything is a component (Summary)

Worksource

WorkSource is a proposal for an open governance, decentralized, peer to peer marketplace for end-users to hire freelancers. It will employ modern cutting edge technology for monetization, accounting, reputation, contractual obligation and in the case needed, arbitration. The most prominent aim is to incorporate very simple access and functionality directly into the CodeDepot userinterfaces. Unlike current freelance market places, freelancers will be made to feel as equals

[45] [@_stream_2017]
[46] [@_compute_2017]
[47] [@_stream_2017-1]
[48] [@_floating-point_2017]
[49] [@_parallel_2017]
[50] [@_pipeline_2017]
[51] [@_cache_2017]
[52] [@_abstraction_2016]
[53] [@_data_2017]
[54] [@_scoreboarding_2016]
[55] [@_direct_2017]
[56] [@_arithmetic_2017]

and not of a second class, as compared to employers. Non-technical users of software often find themselves in need of instruction or in need of customization. By incorporating direct and easy contact with software professionals, the users needs can be addressed painlessly, immediately and reliably.

Technical users find themselves paying exorbitant fees to hire developers at conventional centralized services. Often the employer will find these services confusing, frustrating and simply inadequate.

Freelancers will often find that scanning and applying for jobs, is simply too time consuming.

Current freelance and other work-sourcing like exchanges, are usually run by a centralized corporate entity that enjoy a significant percentage of the cost of the transaction; together with collecting monthly fees. In a decentralized; self-governed; peer-to-peer (P2P) marketplace there isn't any centralized entity, just a community of colleagues and clients. Freelancers enjoy the near entirety of the proceeds of their transactions without some third party dipping their greedy hands in.

New technologies, most significantly bitcoin's blockchain, have now enabled P2P marketplaces to thrive unencumbered by any need of a centralized entity or 3rd party. The need for trust is virtually eliminated.. providing free, flat, P2P markets.

WorkSource will be decentralized, community effort, that will provide reliable sources of service providers to those that need such services. Current cryptocurrency, and other new technology, make it possible to enable most efficient market ecosystems where trust and incentive/disincentive mechanisms are automated; built right in to the platform. This, together with ideas generated and implemented by the community, will make the platform most desirable and efficient to participate in.

CodeSweet

A programmers toolbox is most often burdensome and time consuming to be productive with. It could be argued that every tool in use by engineers is in some way problematic or simply incomplete. If every aspect of every tool and its interface were a component, then the programmer could fashion their tool and hence their toolbox to be just the way they liked it. That bears repeating.. If everything is a component, then the programmer could fashion their tool and hence their toolbox to be just the way they like it!

CodeSweet will be a component based toolkit where engineers have the ability to add features that they deem worthy.. leaving any others behind. Features like automation, intuitive instruction, reimagined user interfaces, and ease-of-use will be of strong focus.

Furthermore

User Interfaces

User interfaces (UIs) are the means by which a user interacts with a computer system. In essence a UI is a portal into the business logic that a software system enables. These portals are also termed the front-end to the business logic's back-end.. a way for a user or software agent to create, read, update or delete data elements. It can be argued that most UIs fall short in their requirements to provide this functionality in a flexible, intuitive and consistent manner.

We often see that UIs often fail with one or some of the following considerations: * Too feature-full.. overburdened by complexity * Context sensitivity woefully lacking in context centricity * Simply unintuitive * Lack of adequate, builtin help systems and wizards. This hinders productivity by being burdensome and time consuming. * Not component orientated which would allow the product to be tailored to the user's needs and desire * No ability to allow the user to color, theme and position interface components as they see fit

User Interfaces, just as software services, should be component orientated. This allows for the user to add and only interact with the features they need. Components would allow for any user interfaces to be customizable and hence personalized. They should also be ubiquotus across all possible platforms.. allowing for software system interaction regardless of which device or platform a user is on. These interactions on differening platform/devices should be consistent and fluid with builtin sychronicity. In summary, a user interface should be intuitive, flexible, and synchronously available on every device.

Other considerations could include builtin feedback or direct access to professional instruction and software customization. Imagine having your own accredited. fairly priced professional just a click away.

Automation

Social Architecture

Economic and Incentive Models

Inflation, Deflation and Distribution

Software

Documentation and Instruction

Computation and Storage

Marketing

User Feedback

Governance

Minimum Viable Product

Priorities (Roadmap)

Infrastructure

Containerization and its Orchestration

- Give a description of what it is and how it is platform agnostic
- Used for blockchain nodes
- Composed of microservices

References