Wirk World

White Paper

22 March 2022

Dr. Robert L. Jones III
Co-founder & Chief AI Engineer, Wirk World, LLP

Overview

Wirk World, LLP is proud to announce the first decentralized autonomous organization (DAO) supporting a novel programming framework for smart agents and smart contracts on the Ethereum blockchain. Most DAOs run on blockchain technology. Ours is no different. We believe the DAO is the future of employment, ownership, leases, contracts, and other business deals. Our DAO ecosystem shall incorporate token-based economics that combines free education, smart contract employment, and the ability to conduct peer-to-peer transactions to accomplish these endeavors. We are offering own cryptocurrency—W1RK tokens—as a means of digitizing assets, peer-to-peer payments, and exchange for fiat currencies—all in an easy-to-use, user-focused distributed application (dapp). Wirk was created to keep pace with global economic growth and to sustainably open the world to employment opportunities, while overcoming international cultural and social differences. This is not about yet another crypto currency or dapp for DeFi or Gaming. Rather, this is about creating a DAO supported with decentralized artificial intelligence (AI) for Good. Good for people. Good for the Planet.

This idea is about providing free education for all with employment upon graduation—anywhere in the World. The challenge we are trying to overcome is that: *where you were born and the country you now live—good or bad—should not prevent you from participating in the global gig economy and the New Information Age.* DAOs such as this is likely the future of careers in the Information Age, one that is trending to decentralization. To that end, the following presents this intriguing idea and our novel solution.

Welcome to Wirk World!

Wirk World Platform

The Wirk World platform has been designed to break through the barriers of entry for those who would otherwise be unable to get an education towards the skills needed to find and keep a good job. Integral to doing this is realizing a multidimensional system solution comprising these key things: the DAO, blockchain technology, and AI—particularly the subfield of machine learning algorithms embedded within and executing in concert with blockchains.

We build AI into Wirk as *smart agents* to guide a client through education and upskill, thereby filling any gaps in their education and ultimately match them with Wirk contracts according to their innate abilities and newfound skills. All these processes are handled autonomously through data-driven and AI-powered smart contracts to provide each user with the best, personalized experience through their education and employment journey.

To explain the Wirk World Platform and our value proposition, we have organized our white paper as follows:

1. Meet your Personal AI Agent
2. Enroll in Free Education
3. Get to Wirk with Smart Contracting
4. Smart Monitoring and Assistance
5. Get Paid with Personal Finance Assistance
6. Refresh and Repeat

Meet your Personal AI Agent

The Wirk World mission is to build a global community of developers, employers, employees, and service providers matched through AI-based interactions and all working together through smart contracts on the blockchain. The innovative combination of AI and blockchain will radically improve the Wirk Life of workers, education for employment, and employers around the World.

Enroll in Free Education

Our training programs are powered by AI to deliver a personalized education plan based on each person’s personality type, aptitude, skills, interests, and user preferences. Our AI *smart agent* is as unique as each user's background and personality. Because of this, we have built our platform with a completely open, transparent business model and blockchain-based systems to keep our organization’s conduct on display for all to examine, keeping our reputation above reproach.

The education will be funded by tokens and fees through smart contracts execution. Saving to do this is also reaped by eliminating bank fees and other third-party expenses that tend to drive up costs. Rather smart contract service fees, positive-energy credits, and the exchange of tokens themselves will generate income that can be channeled to education services.

Get to Wirk with Smart Contracting

Once you have completed your education and skills training and passed your tests—don't fret, the tests are fun—you will be eager to demonstrate to yourself and your personal smart agent how much you've learned.

> `> Yes, learning can be fun, and rewarding. Now it's time to get to work, or as we like to say "wirk".`

Your agent is there now to find employers with jobs already staked with W1RK tokens. Your agent will match you with one or more jobs that are a perfect match for your personal attributes, innate abilities, and your skills.

Smart Monitoring and Assistance

The session with the Wirk agent would be something like this:

> `> Now you're working, ahem, "wirking". How do you like your job? Your employer? Let your personal smart agent know.`
>
> `> Need assistance? That's what your smart agent is for. Having difficulty? Your smart agent can help, and always there to encourage you.`

Ultimately, your smart agent will know when your job is completed to the satisfaction of your employer, automatically, via the smart contract.

Get Paid with Personal Finance Assistance

You got paid in tokens. Now what?

> `> So, you successfully completed your job. Nice job!`
>
> `> Need help using your tokens to buy things?`
>
> `> Exchange tokens for other cryptocurrencies?`
>
> `> Want to exchange your tokens for fiat money?`

How? Your smart agent will show you based on your location, situation, and preferences.

Refresh and Repeat

> `> Now take stock in how you did, and what you learned.`
>
> `> How did this newfound knowledge and skills help you?`
>
> `> Next steps: Keep learning, apply your knowledge and skills.`

```
> Continue your growth to be the best version of yourself.
```

You get the gist.

Programming the Wirk World

Now that we know what Wirk World will do for our clients, how will we program it? This programming task is complicated because Wirk aims to be a distributed application (dapp) on the Ethereum network system supporting all clients anywhere in the world. Indeed, the system will need to accommodate a multiplicity of users with jobs from employers, with differing and even disparate attributes, assumptions, and conditions. So, the Wirk system will need to auto-generate smart contracts (*programs*, albeit from well-defined and verified templates) such that the programs are verifiable. Moreover, our notion of smart agents to assist users requires that AI be an integral part of our system. Incorporating AI into Ethereum (or any other blockchain system) is itself a complex undertaking that has only begun in the blockchain tech community and has not yet been realized to our knowledge. We hope to be leaders in this endeavor and grow a community of developers.

This section will discuss the enabling technologies to be developed:

1. Wirk Framework: Probabilistic Programming
2. Distributed Architecture: System of Generative Flow Networks

Wirk Programming Framework

The Wirk Programming Framework we will develop is shown in Figure 1. This smart contract and AI development framework places formal model-based programming at its core. In so doing we will achieve semi-automatic coding, analysis, verification, learning, and reasoning in one unifying framework supported by a singular model of computation. This *model of computation* intends to support the state-transfer model of the Etherum Virtual Machine (EMV) on which smart contracts execute (from transaction-driven code programmed in the languages of Solidity or Vyper). As a *unifying* model of computation it also supports a transaction-driven dataflow graph model that can be used to program machine learning (ML) algorithms needed to realize the smart agents, program performance models needed to analyze smart contracts, and verify the correctness of the system for testing and auditing purposes.

All together, as a multifaceted hybrid system, the Wirk Framework attempts to realize *computational intelligence*. To do all of that in a world of uncertainty (noisy data and unforeseen events), we require a probabilistic approach formalized mathematically with sound theory and practical solutions.

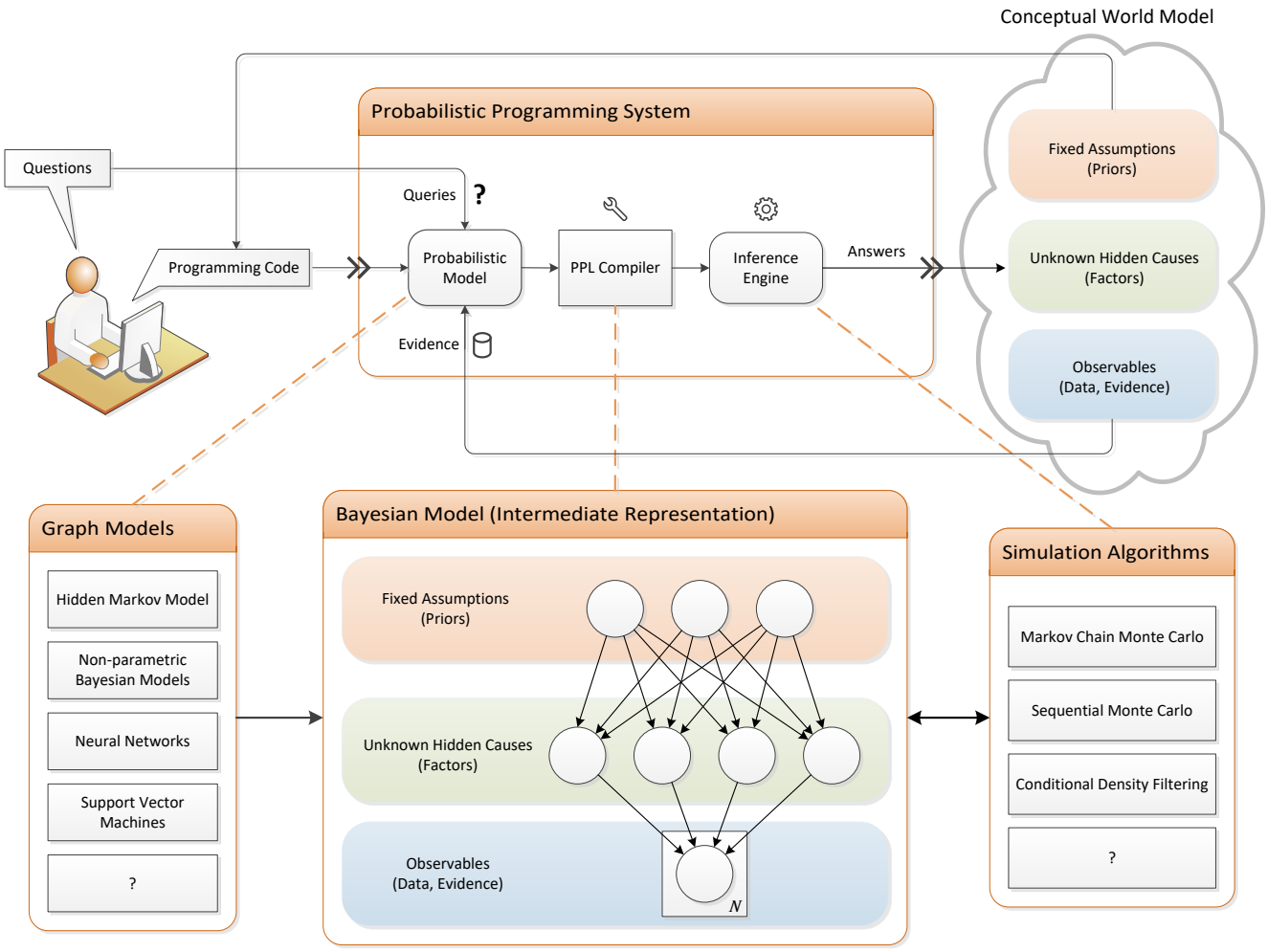

Figure 1: The Wirk programming framework.

Probabilistic Programming

The field of probabilistic programming has grown rapidly in recent years and is now at a sufficient level of maturity [1]. These programming systems now enjoy widespread adoption in real-world applications [2]. But there is more to research and learn in the profiling, testing, verification and debugging of modern distributed computer systems—especially blockchain systems. We propose to leverage the power of probabilistic programming language constructs and compilers to developing smart contracts with embedded smart agents powered by AI processes. We posit that this will enable the programming of smart contract codes that can benefit from both human-programmed and machine-programmed with the aid of ML. The codes will, by design, have a very high-level of modeling power (to describe smart contracts) and sufficient computational power (to execute the code) on the blockchain, hence giving rise to an artificially intelligent distributed network—*computational intelligence*.

This innovation to smart contracts allows the coding of programs that directly represent the data and transaction generation processes in a way that can be modeled, examined via formal

methods, and the execution performance and correctness predicted *before* being deployed and executed on the Ethereum system.

Probabilistic programming, as shown in Figure 2, has recently come into the spotlight and is seen to be as significant to the future of AI/ML as Deep Learning has been of late. A probabilistic programming language (PPL) is a high-level language that makes it easy for a developer to define probability models and then "solve" these models automatically. In their simplest form, PPLs extend a well-specified deterministic programming language with primitive constructs for random choice.

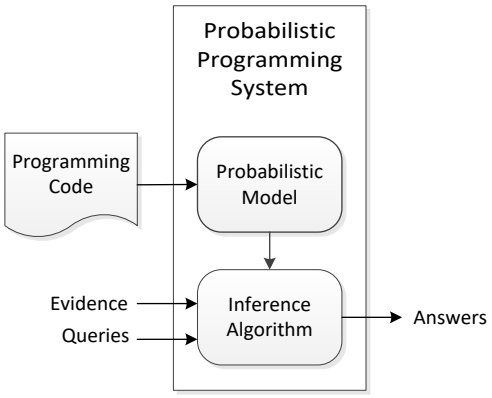

Figure 2: Diagram of a typical probabilistic programming system

A PPL System is either:

language + compiler/interpreter

or

library/framework for an existing language

that

- ✓ includes random choices as native elements,
- ✓ provides a clean separation between probabilistic modeling and inference, and
- ✓ provides automated or semi-automated generation of inference solutions for a given program.

These languages incorporate random events as primitives and their runtime environment handles inference: Now, it is a matter of programming that enables a clean separation between modeling and inference. This can vastly reduce the time and effort associated with implementing new models and understanding data. Just as high-level programming languages transformed developer productivity by abstracting away the details of the processor and memory architecture, probabilistic languages promise to free the developer from the complexities of high-performance probabilistic inference.

Table 1: Key Definitions

General knowledge:	what you know to hold true of your domain in general terms, without considering the details of a particular situation
Probabilistic model:	an encoding of general knowledge about a domain in quantitative, probabilistic terms
Evidence:	specific information you have about a particular situation
Query:	a property of the situation you want to know
Inference:	the process of using a probabilistic model to answer a query given evidence
Representation language:	a language for encoding your knowledge about a domain in a model
Expressive power:	The ability of a representation language to encode various kinds of knowledge in its models
Turing-complete:	A language that can express any computation that can be performed on a digital computer
Probabilistic programming language:	A probabilistic representation language that uses a Turing-complete programming language to representation knowledge

Why Probabilistic Programming?

Probabilistic programming became practical when people realized that inference algorithms that work on simpler representation languages like Bayesian networks can be extended to work on programs. Though the exciting field of PPL is still young, it has grown rapidly, and there are mature software systems available to use; some developed by the author.

PPL compilers typically use Bayesian and Markovian graph models as the Intermediate Representation (IR) to output an inference engine that is executable. Executing the inference engine performs a numerical solution, typically a stochastic solution to a composed Markov Chain Monte Carlo (MCMC) simulation. However it's done, the result is a *generatative model* that can be *sampled* to provide data of a target probability distribution induced by the probabilistic model source coded by the user.

The motivation for probabilistic programming is that it takes two concepts that are very powerful in their own right and puts them together [3]:

Probabilistic reasoning + Turing-complete = probabilistic programming

The result is an easier and more flexible way to use computers to help make decisions under uncertainty.

Probabilistic Reasoning

Probabilistic reasoning is one of the foundational technologies of machine learning. It is used by companies such as Google, Amazon, and Microsoft to make sense of the data available to them. Probabilistic reasoning has been used for applications as diverse as predicting stock prices, recommending movies, diagnosing computers, and detecting cyber intrusions. From the previous section, two things stand out:

1. Probabilistic reasoning can be used to predict the future, infer the past, and learn from the past to predict the future better.
2. Probabilistic programming is probabilistic reasoning using a Turing-complete programming language for representation.

As shown in Figure 2, a probabilistic programming system is, very simply, a probabilistic reasoning system in which the representation language is a programming language. By "programming language" we mean that it has all the features you typically expect in a programming language: data types and structures; variables; functions; control flow.

Consequently, probabilistic programming languages can express an extremely wide variety of probabilistic models and go far beyond most traditional probabilistic reasoning frameworks. In other words, probabilistic programming languages have very large expressive power.

Table 2: PPL System Entities

Entity	How it's defined in a PPL
Probabilistic Model	The probabilistic model is expressed as a program in a probabilistic programming language (PPL) rather than a mathematical model construct.
Evidence	Evidence relates to values of program variables.
Queries	Queries are for values of other program variables.
Inference Algorithm	The PPL system provides a suite of inference algorithms that apply to models written in the language.
Answers	Answers are probabilities of different values of the query variables.

You can use a probabilistic program to predict the future. Just execute the program randomly many times, using what you know about the present as inputs, and observe how many times each output is produced. The magic of probabilistic programming is that it can also be used for all the kinds of probabilistic reasoning using built-in inference algorithms accessed and used appropriately by the language compiler. Not only can it be used to predict the future, but it can

also be used to infer facts that led to outcomes. In other words, you can “unwind” the program to discover the root causes of the outcomes. You can also apply a program in one situation, learn from the outcome, and then use what you’ve learned to make better predictions in the future. So, you can use probabilistic programming to help make all the decisions that can be informed by probabilistic thinking.

There exist probabilistic representation languages with programming language features that are very useful but are not Turing-complete. There also exists simple libs/packages or language constructed embedded in existing languages. But these languages are not Turing complete, so do not carry the same expressive power for building models. We instead focus on Turing-complete language like Stochastic Petri nets [4], SMART [5], [6], Stan [7], the Generative Flow Network [8], and TensorFlow [9], [10].

We believe that the novel contribution we offer is the notion of a “model of computation” that is

1. Powerful—expressible and Turing-complete
2. Constructive—able to generate data objects and code
3. Analyzable—convenient and tractable

On point 1) we can describe anything we want for our smart contracts and smart agents.

On point 2) we can use the same model to produce our smart contracts and smart agents.

On point 3) we can be sure to verify correct performance and guarantee quality of service.

When we combine all three points together, we have an innovate Wirk Framework for programming Wirk World on blockchain for Web3 unlike any other.

Many of the technologies to be used to model, program, analyze, and verify code for Wirk smart agents and smart contracts was developed by the author. The contributions are in both advanced theories and practice through novel software tools [4], [11], [12].

Distributed Architecture

Supporting both Smart Agents and Truly Smart Contracts

The *smart contract* is the immutable program code in the Ethereum system that executes on the Ethereum Virtual Machine (EVM). Transactions are the starting point of every activity in the Ethereum network. Transactions are the inputs that cause the execution of smart contracts on EVMs that, in turn, update variables like balances and, more generally, modify the state of the Ethereum blockchain [13]. The EVM is a state-transition system, a *state machine*, and transactions cause changes in state. Wirk uses the Ethereum blockchain to keep data objects updated per these transactions.

There are two classes of data objects that are *maintained* (created, updated, persisted) either *on-* or *off*-chain:

1. **user-specific data** that is owned by the user client and governed as such, and

2. **user-generic data** that is collected from all users but obfuscated so to be generalized.

Some **user-specific** data is persisted in the blockchain when qualified as part of the global database of record for job training, job qualifications, job contracts in progress or executed, and job performance past and present. The **user-generic** data also persists in a distributed database, but this type of data may be kept on a sidechain or other data buffer and persisted to the blockchain as needed. Both classes of data are used to train the Wirk smart agents to be a personal assistant to each user: a generic agent pretrained from generic data offline (and off-chain) to evolve into better personal agents when instantiated for a new user; an instantiated specific agent continues active learning online (and on-chain) with the benefit for user-specific data. Federated learning is used when data privacy and data movement is not possible otherwise.

In summary and in keeping with Web3 values:

1. Wirk AI should not have a direct access to user-specific data. The user must *allow* the AI use of personal data to train smart agents. So, the process may look like this: User retrieves and decrypts his data from the blockchain and sends it to Wirk AI from the client's side. Certainly, no login/password pairs should be used; rather, only NFT or blockchain address can give an access to the data.
2. User-generic data could be stored anywhere and Wirk AI should not delete it if the user leaves the service (at least according to EU laws).
3. We can guarantee that users own their data only if it is stored on the blockchain. Otherwise, servers could be stopped, and the data could be lost. Technically, we can use a data buffer for better efficiency. Writing data on the blockchain all the time is costly so we may store new data in the traditional encrypted storage and record it on the blockchain periodically (weekly or even monthly).
4. Since users own their data and it is on the blockchain it cannot be deleted. Wirk AI will only be a service that works with federated data provided by user and does not store data itself. The user can close an access to this data anytime and stop using the service. Any user can destroy the key to that data to make sure it can never be accessed in future.

Wirk System

The proposed architecture of our Wirk system is shown in Figure 3. The system is designed to be very modular with well-defined interfaces. On the far left, we employ a multitenancy service for each type of user to accomplish the required tasks. On the far right, the different data modalities must be dealt with to accomplish the tasks. Doing so requires the aid of ML that allows programming the required functions with the data itself. We make the ML solutions modular to leverage any existing commodity software that can be incorporated. Where ML solutions are lacking, we will develop our own using our model of computation presented later and with the help of the TensorFlow framework.

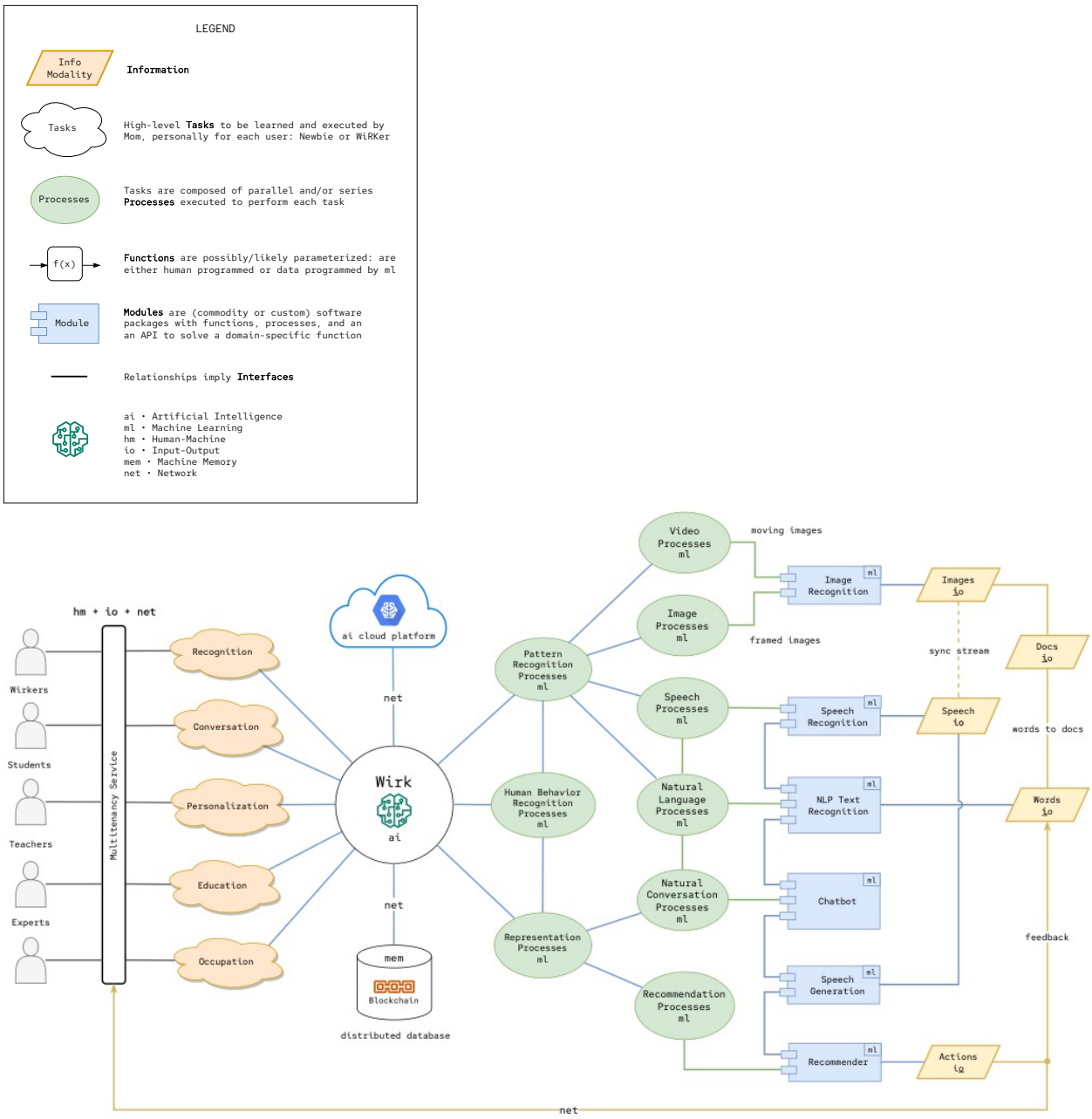

Figure 3: Diagram of the Wirk system architecture to be developed in Phase 1 (first year)

The ML modules support the higher-level processes that are also composable with trainable parameters as well. These processes send and receive messages among these ML modules and generate the data (messages) to perform pattern recognition. To recognize patterns, one must first learn to generate them, hence recognition images and speech, learn to generate speech, text, and recommendations, and realize the smart agent personalized for each client.

At the center of this system is the Wirk AI that will be engineered to orchestrate the processes to accomplish the tasks. To that end, both the AI core and subordinate processes will be designed with the formalisms of graph models: Bayesian networks, Markov chains and networks, and graph neural networks. These design models will be stitched together from the

fabric of a unifying model of computation of generative flow networks. The specifics of this approach are discussed in the next section.

Generative Flow Networks

Nearly all useful models of manmade and natural phenomena are, or can, be conceive as state-transition models. Indeed, nature seems to evolve elegant and optimal designs that tend to maximize the flow itself as the objective (reward) function [14].

For the purposes of crafting and training our ML process models for our smart agents and smart contracts, we use the **generative flow network**, or GFlowNet [15], and abbreviated here as a GFN. This is a new type of generative model trained so that it can sample data objects x (data structures, sets, graphs, code) through a *sequence of constructive steps*. In this context, *flow* refers to the unnormalized probabilities (or energy functions) associated with state transitions decided by functions T: $s \rightarrow s'$ from given a current state, s, to a new state, s'. This model is akin to other graph models with flows: dataflow graphs, Markov chains, Markov networks, queuing networks, and stochastic Petri nets that attach weights to edges between nodes that represent probabilities, energy, cost, or potential functions. And like these other models, there is a notion of *flow-balance conditions* that must be satisfied, usually one that satisfies the Markov property: the future is independent of the past given the present. Finding a solution to these models means finding the settings of the parameters that achieve flow balance.

Each sequence of decisions considers state trajectories $\tau = (s_0, s_1, s_2, \ldots, s_f)$ and actions $(a_0, a_1, \ldots)$ decided by a stochastic policy $\pi(a_t|s_t)$. Each step is a state transition with probability proportional to $R(x)$, some given (or learned) non-negative integrable reward function. This also gives rise to a *stochastic policy* for constructing and sampling this generative process according to *actions.* Actions can represent transactions that not only change the system state, but meant to construct things like plans, designs, contracts, code, analyses, and audits. These things are the data objects used for input and generated as output. This is done in a non-parametric Bayesian formulation where everything is a random variable, and the proper use of prior probability distributions and hierarchical distributions of distributions can be leveraged in the training and inference.

A GFN is a **generative model** because after (and even during) training, we can sample data objects from it [8]. This is better than other generative models for our needs because its training objective is to match a reward function R to the model rather than fitting a dataset. Just as in reinforcement learning, the reward function is what describes the task to be learned—just what we need to learn the tasks the Wirk AI must perform. The GFN generates data objects with probability proportional to a normalized reward function and an inference machine that can answer questions and predict probabilities about some other things, like system state variables—observed or hidden, possible state space, and the most likely state transitions to future states given certain input transactions and events. We can therefore also reason about the possible/likely actions that can take place for a given state trajectory.

The most basic component of a GFN is a **neural network** that defines its constructive policy: i.e., how to sample the next state s_{t+1} given the previous state s_t, through the choice of an action a_t [8]. The new state becomes the input for the next application of the policy, until a special action signals the end of the sequence. This concludes the state sequence and yields a data object $x \leftarrow s_f$ generated from the final state s_f along with a reward $R(x)$ to encourage (or discourage) the policy from generating such a solution—and so encourage more (or less) solutions like it. Being also *stochastic*, this constructive policy is capable of balancing exploitation (of what works well) with exploration (of what may work even better). At any point along the way in this process, we can be assured that each data object x is generated with probability proportional to $R(x)$—a proper probability distribution over x when normalized to sum to 1.

Teaching the Smart Agents

Wirk World will employ subject matter experts and educators to help train the Wirk AI so it can personally engage with, teach, and contract "wirkers" anywhere in the world and guide them on their journey. These are the smart agents that will also monitor the performance of wirkers to encourage, minimize obstacles, overcome shortcomings, maximize success, and verify that the job has been performed according to the smart contract terms.

Optimal Learning

We aim to realize *computational intelligence*—learning, reasoning, predicting, planning, designing, writing—in a way inspired by, and akin to the way humans do with the benefit of our neocortex [16]. To do this, we have, and will continue to build on the theories of optimal learning, Bayesian probabilistic reasoning [17], [18], Markov network model of flows for prediction [3], [7], [10], [15], and generative models.

Reasoning or each plan or each explanation for some given context can be described by a graph obtained by reusing building blocks (causal mechanisms or relations between entities), and conditional GFNs can be trained to sample them or provide probabilistic quantities of interest (conditional probabilities, typically).

Human invention is *sequential* because our brain uses sequential reasoning and learning (what we informally call thinking and predicting as we navigate our environment through our senses). This type of learning takes a lot of time for our human neural network (the neocortex). However, Wirk has been designed to not only learn and evolve iteratively but can do so with the benefit of high-performance computers, manmade algorithms that take shortcuts and approximations, and essentially enjoy accelerated timelines. The Wirk AI will be a distributed, cooperating, and self-organizing smart agents. We will pre-train the smart agent models outside the blockchain with historical data and then deploy the smart agents to process and transact messages. We will also use new data buffered, federated, and later added to the blockchain to continuously

train the deployed agents with a new epoch of training with each new block of transactions and data added to the blockchain.

Our training protocol is based on a combination of **Active Learning** and **Federated Learning** to acquire noisy data samples most useful for training to infer invariant patterns to encode in our ML models.

data samples → example data points

data exemplars → example patterns

Active learning is the selective choosing of data points with highest expected information content for the training set. This is done sequentially using Bayesian model likelihood functions and in the direction of the *knowledge gradient* [20].

Federated learning is a machine learning technique for training models across multiple decentralized nodes (servers or edge devices) where each hold some local dataset for training but none of the data samples are exchanged [21]. In this context, federated learning can be used to train a generic agent model in a decentralized way on local data collected from each user without the data moving.

Think Globally, Compute Locally

We first train the smart agent models to learn from the data generated by active learning for each user. We then use federated learning to update the model parameters across all agents and users.

To sum up, active learning is used to train the smart agents. The agents learn continuously with attention mechanisms focused on user-specific patterns and from user-generic patterns acquired through transfer learning. The combined learning is based on sequential Bayesian learning and inference and is optimal in this context [20]. With this optimal learning, we account for the costs of data acquisition and processing to update the model parameters.

Verification of Smart Contracts

Smart contracts are, traditionally, not very smart. Smart contracts are just code, usually with simple predicate logic and if-this-then-that control flow: when certain conditions are met, the program executes, performs some transaction to some outcome, which includes messages sent to other nodes in the network. The “smartness” is typically limited to make the smart contract code verifiable. We know that we can run an AI/ML program with data on the blockchain itself on any blockchain network with smart contracts because the programming language is Turing complete. With Ethereum, the language Solidity or Vyper is Turing-complete: it can execute any algorithm devised by a human. But now the programming language is too expressive and powerful, and we're now faced with the Halting Problem: the impossibility of verifying in a small finite time whether the smart contract (the code) is correct—

i.e., whether it will be eventually stop before hitting the predefined gas limit and will it perform correctly in the end.

So, the model of computation is usually limited on purpose to a simple state machine (finite automaton), or more powerfully, a pushdown automaton when restricted with certain standards, like ERC-20. This makes the smart contracts analyzable and verifiable by miners. More expressive still, add a second stack, and you can now have loops of instruction streams, yet unroll them from one stack with the help of the second stack to realize more sophisticated contracts, yet still analyze and verify the unrolled loops.

We propose a different approach: to use formal methods—theorem proving methods and exhaustive state-space exploration and analysis methods—to verify these truly smart contracts by formally proving that they are correct. Moreover, this can be done in a tractable way with method proposed here to ensure that mining is efficient (and thus bring down costs by making computation more efficient). These are methods that the author has learned, practiced, and innovated to design, engineer, and verify programs for spacecraft computers. Such software must be proven to be correct before you launch them into space, which makes for a much harder problem than verifying contracts on the blockchain. We can do this! And the technology we will develop to do this is indeed novel and innovative in the context of blockchain.

Model-based Prediction and Verification

Once again, welcome to the world of probabilistic programming languages and frameworks. We use probabilities to describe degrees of belief about the sequence of possible states of transactions on blockchain, the smart agents, and smart contracts. These probabilities are described in a probability graph model where nodes represent random variables and edges connecting nodes represent relationships between the associated random variables. As shown in Figure 4, the relationships are usually causal in nature but can also merely denote a conditional probability distribution. The state of some of these nodes can be observed and known with certainty, others may be hidden from view and must be inferred from the values of other nodes. This is the task of probabilistic inference that allows rational reasoning under uncertainty. We shall use the power of inferential reasoning on the entire state space and state transitions in the blockchain to automatically verify the correctness of the smart contract execution before being sent into the system.

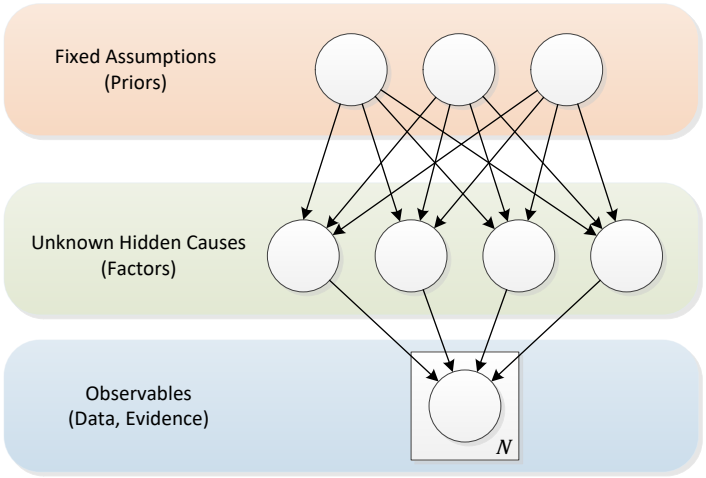

Figure 4: Probability model → generative, graph model → set of data objects (here with cardinality N)

Probabilistic reasoning system require a representation language to express its probabilistic models. The representation language controls what models can be handled by the system via assumptions about independence, dependence, parameters, structures, etc. The set of models that can be represented by a language is called the expressive power of the language. For practical applications, we would like to have as large an expressive power as possible, to avoid the risk of model under-fitting our data (evidence), yet not too much, else we run the risk of over-fitting. A model that is true to the data generation process and fits the data without over/under fitting will likely generalize well to novel (yet unseen) data in the future. With model in hand, there are three basic things that probabilistic reasoning systems do:

1) **Predict** future events
2) **Infer** the cause of events
3) **Learn** from past events to better predict future events.

Model of Computation

Considering the enormity of the state space of very large blockchains, it would be infeasible to construct these graph models manually. Rather we need a high-level language for specifying the possible blockchain evolution, which is subordinate to the execution of many instructions and transactions in the smart agent processes. Such a high-level language would have to include in its model of computation the notion of random variables, the probability distribution functions as well as arithmetic and algebraic operations on the random variables through the associated distribution functions. Then, the language could be used to write down the probability graph model—of a collection of smart contracts subordinated to smart agent processes—as a program itself. In this way, when the program is compiled, a very detailed graph model in the form of a stochastic process called a General State-space Markov Chain (GSSMC) is generated as the low-level model—the low-level machine code, if you will. This machine code is formal and well-defined, so it can be understood by a computer, analyzed,

and even executed to solve the model using numerical methods. The solution algorithms involve exploring and enumerating the state space either exactly with formal methods or through random sampling, usually from MCMC methods.

Our modeling languages of choice comprise the most useful ones that fall into the category of *graph models*. Graph models include Bayesian networks (also known as belief networks) and hidden Markov models (HMMs). The Markov chain is exactly a probabilistic state machine that can model the construction of a blockchain. The (stochastic, extended) Petri net (PN) is Turing-complete and can, therefore, model the execution of smart contracts on the EVM [22]. The stochastic PN (SPN) model emits an underlying GSSMC, which is equivalently a model of a discrete-event simulation. A reachability graph (RG) of a PN is a directed graph $G = (V, E)$, where $v \in V$ is the set of reachable markings (PN states), and $e \in E$ represents the directed edges that connect markings, however large, that can be analyzed and checked for desired properties, which is used for formal verification. If the PN is said to be "timed", this means that delays are attached to transitions that cause the PN to sojourn in states for some deterministic or random time. We then call the PN a *stochastic* Petri Net (SPN). The underlying reachability graph of a SPN is a GSSMC as the notion of "state" now includes the *age* of the PN marking, or equivalently, the *remaining time* until a transition fires, thus causing a change to a new state [4].

Indeed, for useful modeling formalisms, we have many choices. So, we need a **model of computation** as an intermediate representation that is expressive, thus powerful, yet convenient. For this we choose the aforementioned Generative Flow Network (GFN). It checks all the boxes. Though we can construct a GFN model manually when special crafting is warranted to leverage special knowledge being captured, the easier way is to use a high-level language to generate the low-level GFN. We propose to develop the toolchain shown in Figure 5 that will use the languages of Python and stochastic Petri nets for the good reasons discussed here. In subsequent development phases we might even consider creating our own compiler for Solidity, Vyper, or other to translate the source code of smart contracts directly to GFN models. But, for now, we will be content with Python and Petri nets.

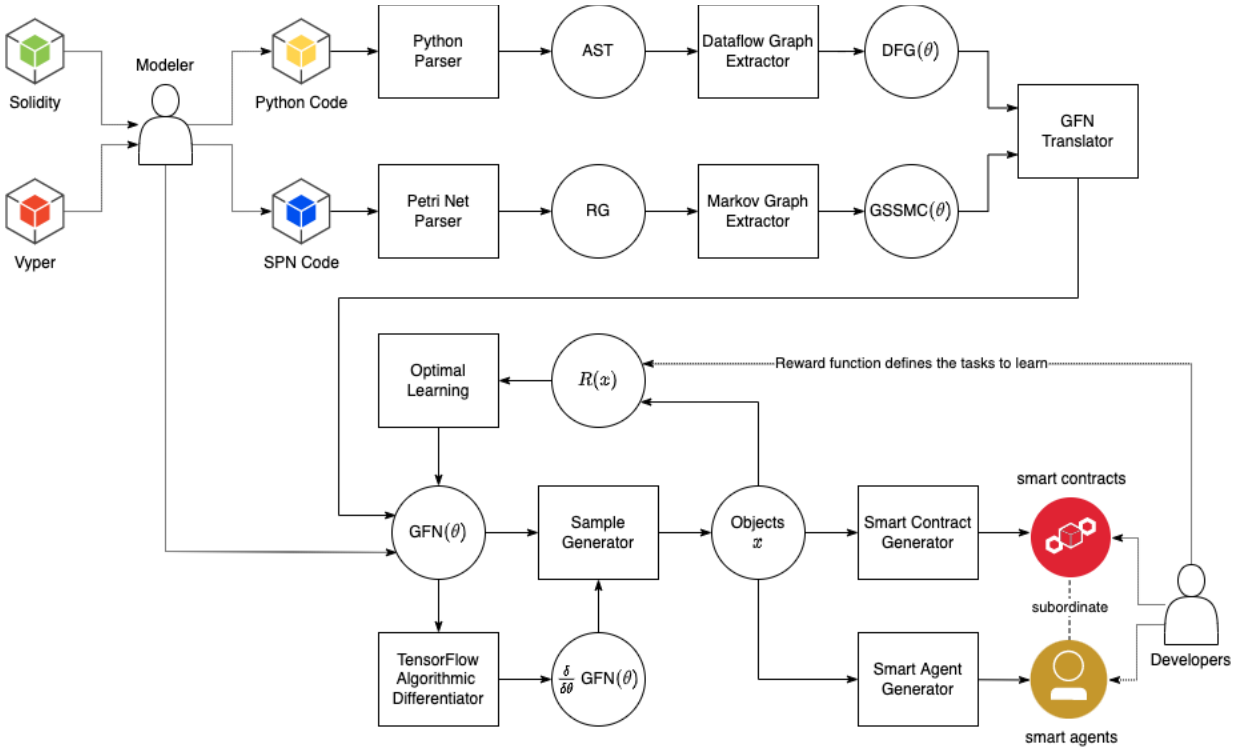

Figure 5: Wirk Framework Toolchain

As the toolchain in the figure above shows, we have replaced the classical MCMC sampler by a trained GFN model—the advantage being that the GFN can exploit the $(x, R(x))$ pairs seen during training sequences and then use that information to further generalize matters elsewhere in the process [8]. That is to say, the GFN generalizes from seen (statistics) modes of the unnormalized probability (energy) function to new modes, and then jump directly to those modes by simply *sampling* from the GFN. In contrast, the MCMC is disadvantaged in that it must stochastically *search* for new modes by making lots of small perturbations in its random walk of the state space. The GFN way is better, faster, cheaper.

There is much related work in the literature on the topic of formal verification of smart contracts [23]. The approaches tend to fit in one of two classes, either contract-level or program-level. Our approach falls in the contract-level camp because we intend to model the state-transition behavior of a given contract and as affected by transactions. This can be easily modeled by Python/SPNs (source) and GFNs (model of computation) by gleaning the salient features of behavior from the smart contract code (Figure 1 and Figure 5).

Program-level approaches to verification instead parse the source code itself (Solidity or Vyper) and emit the model of computation directly for analysis and verification. This would require us to develop our own compiler. While this is could be done with LLVM [24], as used by other modern languages, we think that contract-level modeling is a good start—considering everything else there is to do to build the Wirk system and dapp (see Roadmap). As we alluded to above, subsequent development phases may revisit this topic and consider a new compiler technology. Perhaps one of the works sited in [23] will provide a compiler for us in the future.

For now, we feel that our approach is more than sufficient considering that there is little to no tools available now for this kind of formal verification. Indeed, some of the related work also uses Petri nets, though *colored* Petri nets. But colored Petri nets are just a modeling convenience, as the extended stochastic Petri net we use is already Turing-complete—so no added modeling power can be had by coloring the tokens. We believe our approach presented here is novel as well as innovative in its use of generative flow networks as a model of computation. This affords us many benefits, as already stated, and soon to be leveraged in our development Roadmap presented at the conclusion of this white paper.

Computational Efficiency

As to the practicalities of turning the thousands of EVMs, affording the gas is a challenge. It could use a lot of gas if not done properly, and gas is expensive on Ethereum. This challenge has occupied our time of late. The gas prices of Ethereum could prohibit any AI/ML process on the blockchain itself. We propose to side-step this problem in at least two ways: 1) the use of side chains to execute the smart agents in a way that is more cost-effective and 2) with self-sustaining bio-gas systems and credits for energy-efficient contracts to reduce our carbon footprint.

We propose to execute the smart agents in a sidechain that runs as Layer 2 (in parallel and pegged to Mainnet) where the execution of the smart agents is more efficient and does not burn the expensive gas; but rather our own fuel in W1RK tokens and priced cheaper. The whole point to blockchain networks, as we know, is to incentivize users and miners to take part in operating and maintaining this distributed network of computers and device clients at the edge. We also want to incentivize developers like us to write efficient and verifiable code, to make efficient use of memory and bandwidth, and want them to perform mining so that we have blocks to run with. How do we get cheaper fuel? We include in the mix the bio-gas system: the same theory as hydropower applies, but for bio-power in service agreement with SEM.world [25] That would make fuel costs negligible (nearly nonexistent) once the asset is paid off. We are just setting one plant, and everyone would want it.

Roadmap

The road ahead to develop Wirk World has many challenges. We hope that the approach presented here strongly suggests that it is not only possible, but practical and worthwhile for the reasons discussed. This Roadmap to the future can be summarized around a two-phased development plan, which is discussed in the remaining sections:

1. First, **centralized smart agents** for decentralized smart contracts (Figure 6)
2. Then, **decentralized smart agents** so that everything is distributed (Figure 8)

We intend to design and develop our solutions to evolve iteratively with agile software engineering practices. The development plan for engineering and scientific staffing is diagramed in Figure 9.

Phase 1: Centralized Smart Agents

In the first year, we will build the Wirk system according to the architecture diagram in Figure 1. We will build on the Ethereum blockchain to hold the distributed database of transactions between users: newbies (being educated), the educators (both for humans and the smart agents), wirkers (educated and employed), and the Wirk employers. The blockchain would store all salient states and data that over the Wirk Life Cycle. This is portrayed in Figure 6 where the nodes are defined as:

Full node

- Stores full blockchain data so all states can be derived from full node.
- Participates in block validation, verifies all blocks and states.
- Serves the network and provides data on request.

Light node

- Stores the header chain and requests everything else.
- Can verify the validity of the data against the state roots in the block headers.
- Useful for low-capacity devices, such as embedded devices or mobile phones, which cannot afford to store all the blockchain data.

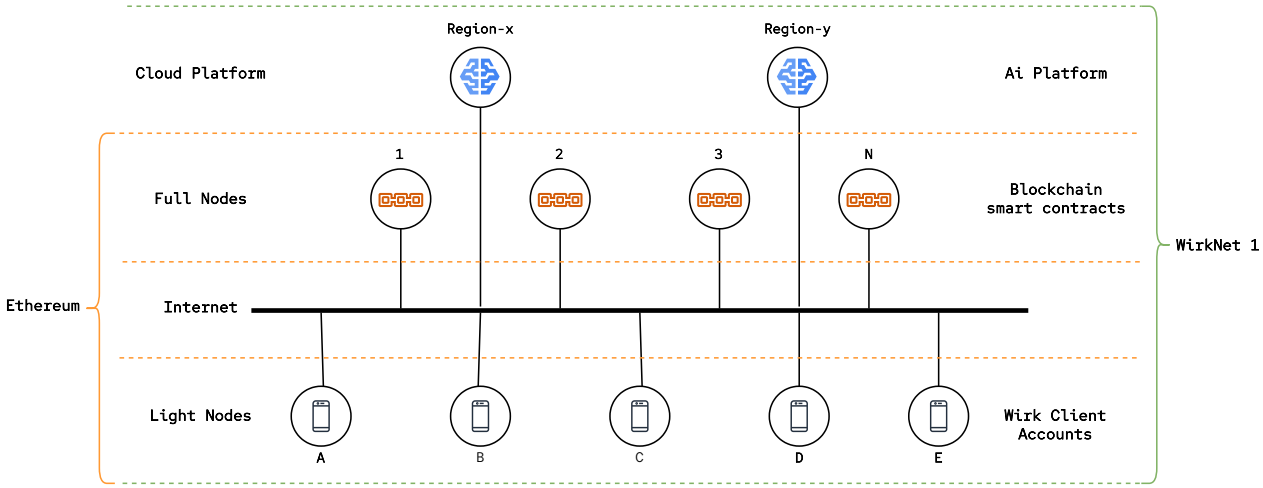

*Figure 6: Wirk will be realized in **Phase 1** development to operate with **centralized** AI*

Where our distributed database and ledger of client transactions with educators and employers, payment transactions—the entire state transition history there for transparency, auditing, and performance monitoring of all Wirk processes among all participants: students, educators, employers, and wirkers alike.

The GFN is our model of computation of the state-transitions initiated by transactions for both the smart contracts and our smart agents embedded in the Ethereum system, either through oracles to the smart agents or embedded at the edge or in a sidechain.

Goals

- Build the Wirk World dapp on the Ethereum blockchain.
- The blockchain will be used for transactional data that should be stored safely to persist the unalterable history of all transactions.
- The Wirk AI agents will be engineered and realized in the Google Cloud using TensorFlow and supporting Vertex AI resources [26]. This will afford us the cutting-edge tech and the least technical and schedule risk towards achieving our goals in Year 1.

Objectives

- Develop a PPL compiler for Python, leveraging Python's:
    - Convenient access to the AST to craft our own syntax and semantics on top
    - Interpreter and dynamic data typing in generating run-time code
    - Python generator construct and function closures with decorator syntax
    - Ease at which we can create interfaces to other languages through wrappers
- Develop a Python wrapper for the stochastic Petri net language.
- Develop a GFN model of computation—a fabric to stitch together our system of computational solutions.
- At first, make use of existing MCMC inference engines—such as Stan—to more rapidly compose and test our GFN-based models emitted from our new PPL compiler.
- Revise the PPL compiler and inference engine to replace the MCMC sampler with our GFN model sampler for the advantages stated above.

Remarks: The above will make it easier for our Wirk Framework and PPL compiler to construct the inference engine to generate samples from the target posterior distributions defined expressively over the observable data from the input space; the model parameters/hyperparameters; and the hidden/latent variables—any or all the same. The same model of computation would also serve emit a performance model and verification model to validate the smart agents and smart contracts before putting into service. Moreover:

- Using the above Wirk Framework, design and code the smart contract templates (to seed the auto-generator feature) that can perform all the needed tasks in the Wirk Life Cycle.
- Develop the smart agents using the GFN model of computation with the language and run-time system of TensorFlow. This involves adapting existing ML solutions through APIs, adapting existing TensorFlow solutions for the same, or coding new TensorFlow solutions where needed.
- With the help of Vertex AI, develop Wirk AI of smart agents with GFN models in TensorFlow that makes use of the ML solutions above as procedures and functions.
- Design and implement the AI training protocol and reward functions for the GFN models so that human SMEs can teach the Wirk AI through active learning and reinforcement.
- Test, demonstrate, improve the design in the first year of Phase 1.

Phase 2: Decentralized Smart Agents

In Phase 2 development, as shown in Figure 7, we will migrate the smart agents from being centralized to a decentralized system with embedded ML. This will distribute the smart agents with ML solutions that are both embedded at the edge and in a sidechain. In so doing, Wirk AI will be able to leverage the thousands of EMV nodes into one large, distributed virtual computer without the cloud service. Enlisting the entire Ethereum blockchain gives rise to a supercomputer, distributed, tamper proof, controlled by none, and benefiting all.

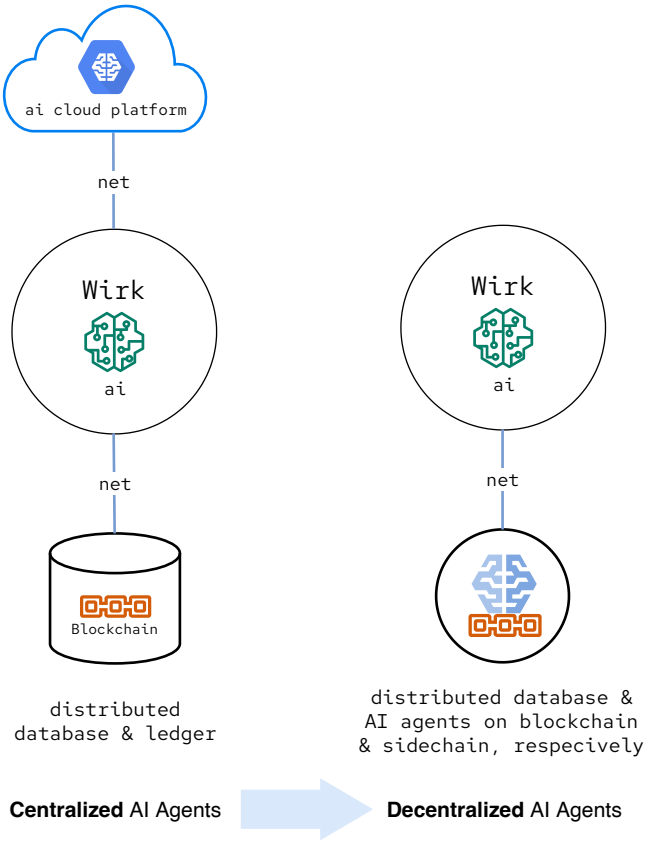

Figure 7: Changes to architecture from Phase 1 to Phase 2

In addition to the Full and Light nodes, we add something new: smart agent nodes with ML functions at the edge, or Edge ML. This is portrayed in Figure 8 where the added node type is:

Edge ML node

- Same as Light node.
- Also has embedded, distributed AI functions to service each client.

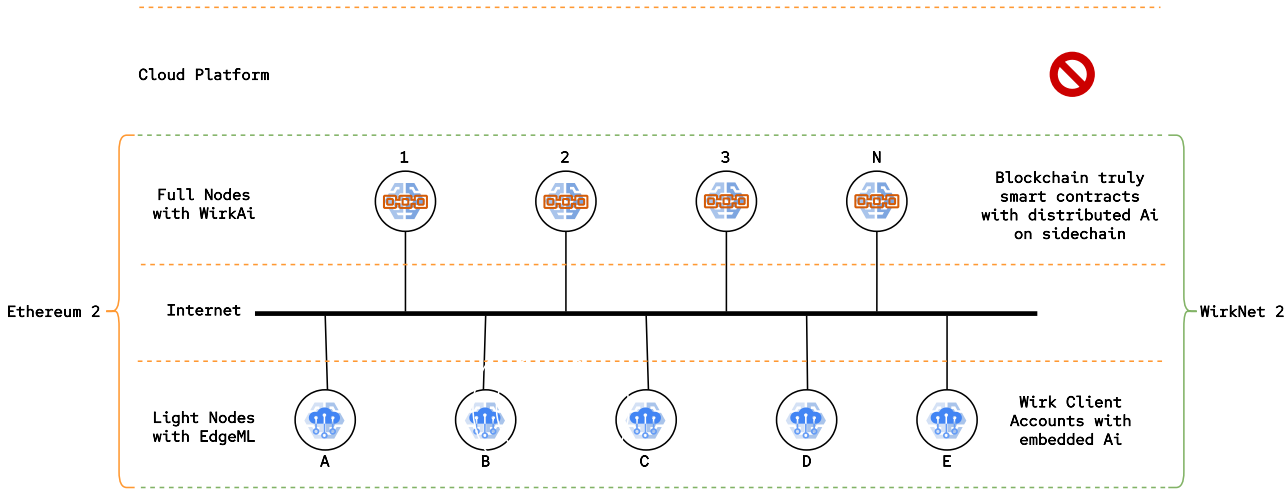

*Figure 8: Wirk will be realized in **Phase 2** development to operate with **decentralized** AI*

Goals

- Capitalize on what we engineered in Phase 1 and make quality-of-service improvements for an enhanced user experience. These improvements will also further reduce costs and maximize throughput (minimize latency) using a sidechain designed to allow the centralized smart agents to be distributed across the Ethereum network as shown in Figure 7.
- Decentralize the Wirk AI smart agents to the edge to allow distributed execution alongside the blockchain itself.
- To that end, translate the TensorFlow models developed in Phase 1 to TensorFlow Lite [27].
- Develop a sidechain as an independent EVM-compatible blockchain that runs in parallel to, and compatible with the Mainnet.
    - Like any sidechain, communication with the Mainnet is via two-way bridges.
    - Our sidechain will run under its own rules of consensus and block parameters—under the policy and execution of our model of computation realized by the GFNs—and, at its terminal conclusion, the resulting data objects will be persisted to the blockchain on Mainnet.
    - For efficiency reasons, these terminal transactions and data objects can be buffered and submitted to the Mainnet in a lazy way over spanning a longer period to alleviate Mainnet congestion and keep costs lower.

Objectives

- Design and develop the distributed, cooperating, and self-organizing smart agents.
- TensorFlow is a beast. We cannot readily put that dependency and burden on our smart agents or smart contracts, especially on EVMs. Rather, there is TensorFlow Lite for deploying previously trained models on a bare bones, light-weight, highly-

optimized version designed as a run-time system for embedded ML functions at the edge [27]. This is ideal and a proven technology supporting smart phones, tablets, and other resource-limited devices. This path to embedded distributed ML is supported by the languages used to program apps on these devices [10]. We will translate our previously trained GFN-based TensorFlow models to TensorFlow Lite, integrate, test, and repeat until optimized.

- Investigate the use off-chain Layer 2 scaling strategies—rollups and state channels—to allow the smart agents to interact with the Ethereum system more efficiently, with higher throughput, and low latencies—all needed for the best user experience. Off-chain solutions are implemented separately from Layer 1 Mainnet, as they require no changes to the existing Ethereum protocol. Some solutions, known as Layer 2 solutions, derive their security directly from Layer 1 Ethereum consensus. These solutions communicate with the Mainnet but derive their security differently to obtain a variety of goals.
    - Rollups perform transaction execution outside layer 1 and then the data is posted to layer 1 where consensus is reached. As transaction data is included in layer 1 blocks, this allows rollups to be secured by native Ethereum security.
    - State channels utilize multisig contracts to enable participants to transact quickly and freely off-chain, then settle finality with the Mainnet. This minimizes network congestion, fees, and delays. The two types of channels are currently state channels and payment channels.
- Reduce the cost of transactions and maximize throughput by implementing Layer 2 rollup strategies that scale up efficiently as an Ethereum-based system.
- Design of fuel tokens and protocols for the Wirk AI sidechain.
- Optimize and scale up the system.

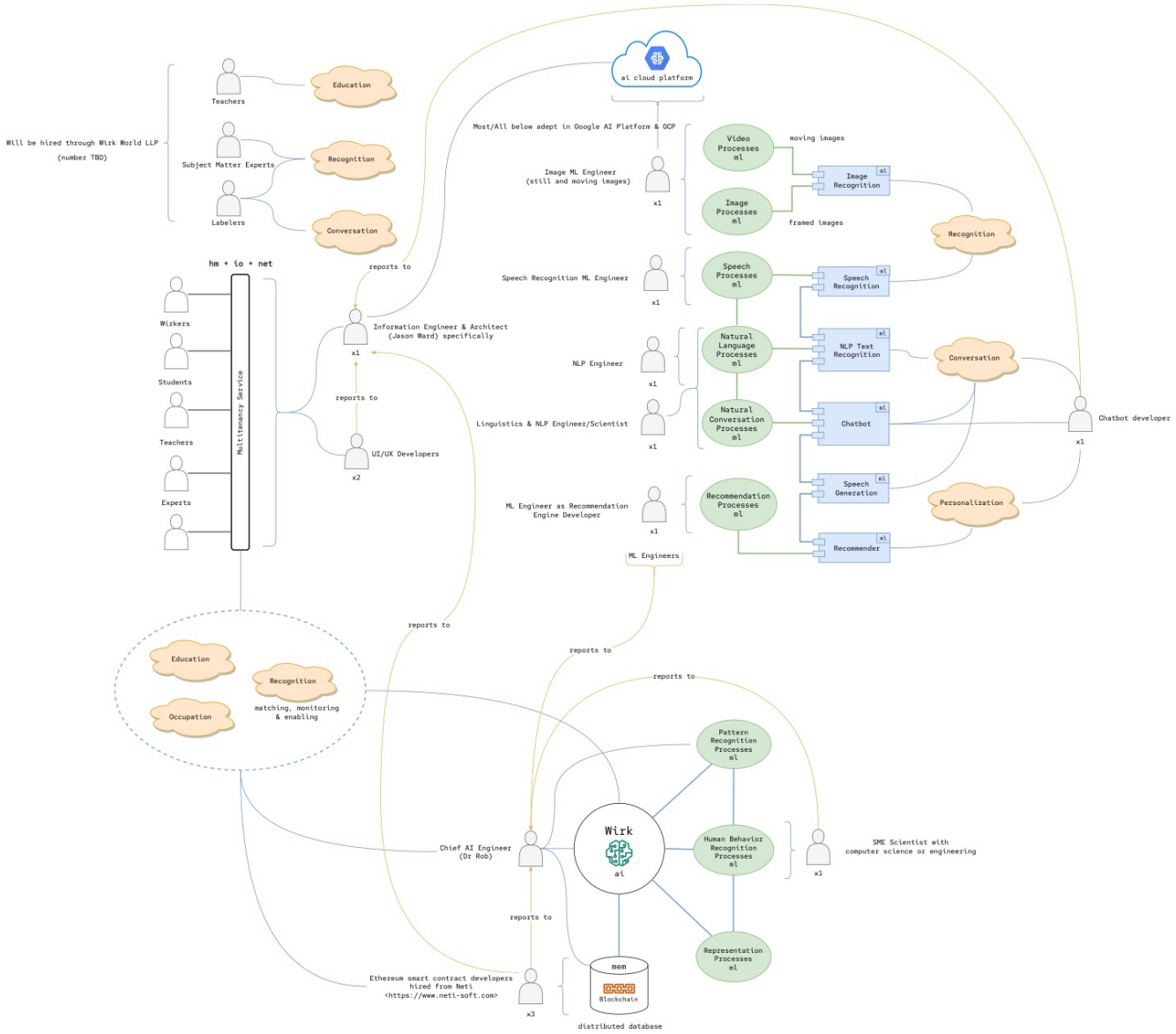

Figure 9: Diagram of entity relationships to staffing

References

[1] B. Cronin, “What is probabilistic programming? - O’Reilly Radar.” http://radar.oreilly.com/2013/04/probabilistic-programming.html (accessed Feb. 12, 2015).

[2] Beau Cronin, “Probabilistic Programming: Why, What, How, When?,” 20:58:54 UTC. Accessed: Feb. 09, 2015. [Online]. Available: http://www.slideshare.net/salesforceeng/probabalistic-programming-why-what-how-when

[3] Avi Pfeffer, *Practical Probabilistic Programming*. 2015. [Online]. Available: http://www.manning.com/pfeffer/

[4] Robert L Jones III, “SIMULATION AND NUMERICAL SOLUTION OF STOCHASTIC PETRI NETS WITH DISCRETE AND CONTINUOUS TIMING,” College of William & Mary, Williamsburg, VA, 2002.

[5] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu, “Logical and Stochastic Modeling with Smart,” in *Computer Performance Evaluation. Modelling Techniques and Tools*, P. Kemper and W. H. Sanders, Eds. Springer Berlin Heidelberg, 2003, pp. 78–97. Accessed: Mar. 05, 2015. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-45232-4_6

[6] G. Ciardo, R. L. Jones, R. M. Marmorstein, A. S. Miner, and R. Siminiceanu, “SMART: stochastic model-checking analyzer for reliability and timing,” in *Proceedings International Conference on Dependable Systems and Networks*, Jun. 2002, pp. 545-. doi: 10.1109/DSN.2002.1028976.

[7] Stan Development Team, *Stan Modeling Language Users Guide and Reference Manual, Version 2.5.0*. 2014. [Online]. Available: http://mc-stan.org/

[8] Y. Bengio, T. Deleu, E. J. Hu, S. Lahlou, M. Tiwari, and E. Bengio, “GFlowNet Foundations,” *ArXiv211109266 Cs Stat*, Nov. 2021, Accessed: Mar. 15, 2022. [Online]. Available: http://arxiv.org/abs/2111.09266

[9] “TensorFlow,” *TensorFlow*. https://www.tensorflow.org/mobile/ (accessed Feb. 24, 2017).

[10] “TensorFlow Lite API Reference.” https://www.tensorflow.org/lite/api_docs (accessed Mar. 20, 2022).

[11] Robert L. Jones III, “Self-organizing sequential memory pattern machine and reinforcement learning method,” US8504493B2, Aug. 06, 2013 Accessed: Mar. 07, 2022. [Online]. Available: https://patents.google.com/patent/US8504493/en

[12] R. Jones and R. F. Hodson, “Application of RSC to Spaceborne Lidar Data Systems,” p. 11.

[13] “Mastering Ethereum [Book].” https://www.oreilly.com/library/view/mastering-ethereum/9781491971932/ (accessed Mar. 19, 2022).

[14] A. Bejan and J. P. Zane, *Design in Nature: How the Constructal Law Governs Evolution in Biology, Physics, Technology, and Social Organizations*, Illustrated edition. New York: Anchor, 2013.

[15] E. Bengio, M. Jain, M. Korablyov, D. Precup, and Y. Bengio, “Flow Network based Generative Models for Non-Iterative Diverse Candidate Generation,” Jun. 2021, doi: 10.48550/arXiv.2106.04399.

[16] J. Hawkins and S. Blakeslee, *On Intelligence*. Macmillan, 2007.

[17] Judea Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. 1988. [Online]. Available: http://www.amazon.com/Probabilistic-Reasoning-Intelligent-Systems-Plausible/dp/1558604790

[18] J. Pearl and D. Mackenzie, *The Book of Why: The New Science of Cause and Effect*, 1 edition. New York: Basic Books, 2018.

[19] Y. Bengio, “The Consciousness Prior,” *ArXiv170908568 Cs Stat*, Sep. 2017, Accessed: Apr. 12, 2019. [Online]. Available: http://arxiv.org/abs/1709.08568

[20] W. B. Powell and I. O. Ryzhov, *Optimal Learning*. John Wiley & Sons, 2013.

[21] P. Kairouz *et al.*, “Advances and Open Problems in Federated Learning,” *ArXiv191204977 Cs Stat*, Dec. 2019, Accessed: Jan. 04, 2021. [Online]. Available: http://arxiv.org/abs/1912.04977

[22] “Robert L. Jones, Analysis of Phase-Type Stochastic Petri Nets With Discrete and Continuous Timing, NASA/CR-2000-210296, November 2000, pp. 91.”

[23] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, “A Survey of Smart Contract Formal Specification and Verification,” *ArXiv200802712 Cs*, Apr. 2021, Accessed: Mar. 16, 2022. [Online]. Available: http://arxiv.org/abs/2008.02712

[24] “The LLVM Compiler Infrastructure Project.” https://llvm.org/ (accessed Mar. 22, 2022).

[25] “SEM.” https://sem.world/ (accessed Mar. 07, 2022).

[26] “Vertex AI,” *Google Cloud*. https://cloud.google.com/vertex-ai (accessed Mar. 20, 2022).

[27] “TensorFlow Lite | ML for Mobile and Edge Devices,” *TensorFlow*. https://www.tensorflow.org/lite (accessed Mar. 22, 2022).

