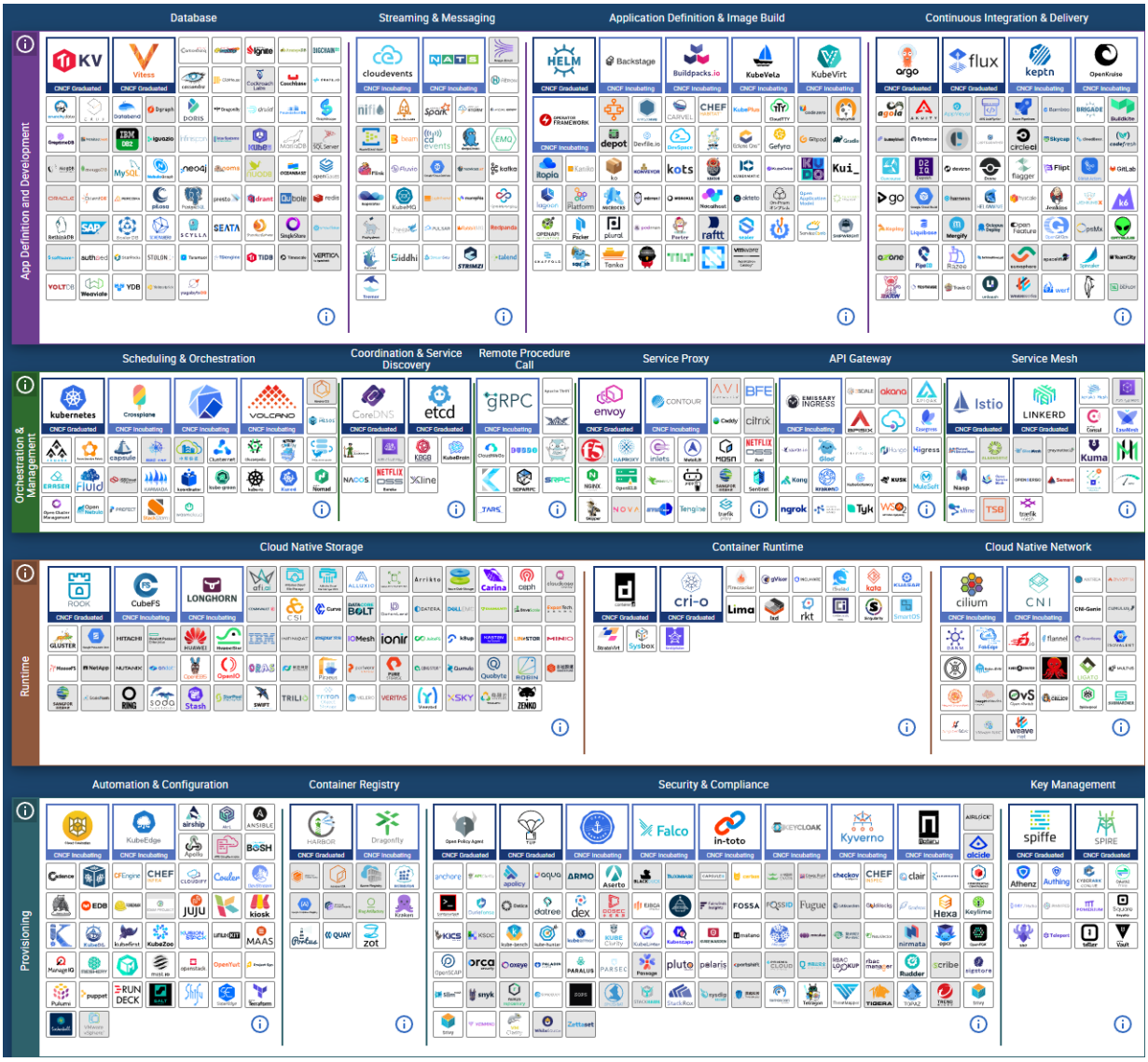
Introduction To Virtuoso 1.0.0

Background

This document provides a comprehensive introduction to Virtuoso and Montage to capture all of the relevant background and useful conceptualizations. Comprehensivenses, rather than conciseness, is the main goal, because as users become more aware of the multifaceted problems that it solves and the power that it provides, the more users will naturally invest in using the infrastructure and building out content.

Software technologies, like music genres or languages, continue to morph and evolve. Some technologies share a close common evolutionary ancestry, while others are more distant. Different technology domains, like embedded, cloud, simulation, etc., develop their own distinct dialects, sometimes reproducing the same concepts. The cloud-native landscape shown below, from the Cloud Native Computing Foundation, gives an idea of the extent of this proliferation of ideas.



Each of these technologies tells a story about a problem that was identified, and a solution created to solve it. This introduction to Virtuoso and Montage is intended to provide readers with context for clearly understanding the problems that Virtuoso and Montage solve and where they fit in the constantly evolving technology landscape.

The Golden Path

Spotify's development team had been using DevOps, the most widely adapted software development process of the time (particularly for cloud native technolgies), and noticed that new problems emerged. Developer productivity was challenged by the continuously increasing complexity of their DevOps processes, which created high cognitive load and frictional losses from context switching. To solve this problem, they developed an internal platform to allow their developers to work more productively. They saw this platform as needed to create a "Golden Path" of software delivery. The Golden Path represents the optimum path that development teams can take, from product concept to delivery, of all possible paths. Spotify also recognized this internal development platform represented a core function that solved critical problems not just for their organization, but for organizations at large. So they released the infrastructure they developed as a product called Backstage, which helped solidify platform engineering as a concept.

Backstage was released as a generalized solution to the specific problems they faced in the development and maintenance of their core software products. Spotify's development team had the clarity to understand the need for the Golden Path as a concept for their company, and recognized that while the specifics of what a Golden Path are will be different for every company, there was enough overlap in Golden Paths to justify a generalized infrastructure. Platform engineering identiifies a common need for self-service, orchestration, and organization, focused on developer productivity, where the developer is the end customer/user. Platform engineering itself is the practice of using dedicated teams to create custom internal development platforms ("IDPs") to maximize developer productivity. According to Gartner, platform engineering is "an emerging technology approach that can accelerate the delivery of applications and the pace at which they produce business value. Platform engineering improves developer experience and productivity by providing self-service capabilities with automated infrastructure operations. Platform engineering is trending because of its promise to optimize the developer experience and accelerate product teams' delivery of customer value."

Virtuoso shares strong common roots with the platform engineering evolutionary tree, with a clear focus on the importance of the Golden Path. We use the concept of differentiation to add some richness to robust understandings of the Golden Path.

Ontogeny: The Differentiation Process

For all organizations, the Golden Path ends with a clear organizational objective. For Spotify, that objective is the creation of music and digital content streaming services. The diversity of ways that the Golden Path ends for companies is unlimited, ranging from defense systems to medical devices to consumer electronics. We could imagine each of these end software applications as distinct animal species as varied as life itself, each fully formed animal represents the differentiated "thing" engineering teams set out to build.

Every animal alive was built from a single fertilized egg cell, the zygote. From that one cell, all of the different cells were grown and organized to manifest a fully differentiated animal. This is the process of ontogeny, and represents an intuitive analogy to the software development process. It begins with the zygote, a single cell with unlimited flexibility that represents the foundational programming languages used to create any software imaginable. The process of software ontogeny, then, directly maps to the concept of the Golden Path, so it's worthwhile to examine the insights found in ontogeny.

The zygote's journey from one cell to trillions of diverse, perfectly related skin cells, muscle cells, and brain cells, etc., involves a process of differentiation. The zygote contniues to divide into omnipotent stem cells, which can literally become anything. These further divide into slightly more specialized pluripotent stem cells, which can become many things. These then divide into the myriad different specific cell types. What we can see then, is that in the process of differentiation represents a process of creating cells that are **more useful** for **specific functions**, but also **less flexible**. In other words, in the search for utility, there is a general tendency to sacrifice flexibility. Importantly, there is no going back. The process is one-way: once differentiation has occured, flexibility is lost.

Software teams use shared libraries and frameworks, so of course software application design is supported by more than just the programming language. But the build process involves a leap from source code and libraries to fully formed application, a jump from zygote straight to fully differentiated animal, without an intermediate "pre-differentiated" stem cell stage of composition, design, and maintenance. A key role of the Virtuoso infrastructure is to expand upon the differentiation process and highlight the importance of ontogenic flexibility in the creation of radically powerful Golden Path possibilities. As no-code infrastructure, Virtuoso represents a dramatic expansion of a "pre-differentiated" stage of development between zygote, or source code, and fully differentiated animal, or end software solution. However, there is another dimensions of flexibility key to the creation of Golden Paths in addition to ontogenic flexibility, which we call metabolic flexibility.

Metabolic Flexibility

Organisms require energy to function, just like software applications need capabilities to function. To get this energy, organisms must eat macronutrients: carbohydrates, fats, and proteins. But there are significant differences in how organisms metabolize food, even within the same species. One person to the next will have different functioning of metabolic pathways, based on genetics or environmental or lifestyle factors. One key difference is metabolic flexibility, the ability of a person to efficiently switch between burning different macronutrients to power life processes. A metabolically inflexible person may have difficulty burning fat, or be insulin resistant and have difficulty metabolizing carbs. Research has shown that metabolic inflexibility is a key contributor to all-cause mortality. Cellular infrastructure is needed to maintain metabolic flexibility to efficiently burn fat, and this infrastructure is mitochondria. But like any infrastructure, this cellular infrastructure needs to be built. In biology, this can be accomplished in various ways, including exercise, intermittent fasting, and cold exposure.

In software, metabolic inflexibility describes a general inability to compose pre-existing capabilities together to form a solution. A distinctive rigidity that creates fault lines between capabilities that either limits usability or involves substantial implementation complexity. Monolithic software systems that exhibit strong metabolic inflexibility are characteristically brittle, not easily adaptable, and often difficult to maintain.

In contrast, a metabolically flexible software development environment follows Gartner's Composable Business Software Architecture. A composable architecture enables rapid change, and rapid change can easily be the difference between life and death for enterprises experiencing a changing business environment. The Golden Path means more than simply building the software and calling it done. The Golden Path means building a living software organism that is easy to maintain, scale, and manage.

Per Gartner, the four pillars of composable software are modularity, discoverability, orchestration, and autonomy. With the foundational concepts presented so far, we are able to begin describing the specific functions of the Virtuoso and Montage infrastructures. The detailed introduction could begin with either one, but we present the Montage infrastructure first because, as we will see, Montage is less "opinionated".

In The Beginning, There Were Packages

Virtuoso represents general purpose composable no-code infrastructure designed to enable optimum Golden Paths for a variety of workflows. Virtuoso also represents a digital ecosystem with complex requirements that are shared by other ecosystems. During the development of Virtuoso, it became overwhelmingly apparent that Virtuoso, as a digital ecosystem, had infrastructure requirements that were shared by other digital ecosystems. As a result, Montage was built as a separate infrastructure. It is designed to provide general-purpose modularity, discoverability, and orchestration for two-sided digital ecosystems and, crucially, also provide digital economy infrastructure. The first foundational concept of Montage is the generic notion of packages and ecosystems. The Montage infrastructure is not understood until the value of generic notions of packages and ecoysstems is understood.

Montage consists of a package technology that is explicitly designed to be adaptable to any ecosystem to support the concept of "packaged capabilities" in Gartner's composable business architecture. Some characteristics of Montage packages:

Montage packages should be compatible with any existing package technology (Nuget, Maven, pip, ec.)
Montage packages are laid out as contents of a root folder with three subfolders.
The "Payload" folder in this root folder contains any content to be governed, orchestrated, and notionally delivered or installed.
The "Montage" folder contains package metadata used by Montage in the orchestration of the package. This layer of orchestration is general purpose orchestration in common with all digital ecosystems.
The "Ecosystem" folder contains package metadata used by the ecosystem's extension of Montage for orchestration of the package.
Packages form dependencies on other packages
Packages can belong to one or more ecosystems
Packages evolve over time, and are versioned
Package version **notations** can vary from ecosystem to ecosystem
Package version **dependency notations** can vary from ecosystem to ecosystem
How packages are installed varies from ecosystem to ecosystem
Governance must be applied to packages at the time of orchestration

The last point is perhaps key to understanding the necessity of Montage as an infrastructure layer. Teams setting out to create a Golden Path will fail if they have a so-called "Day 1" infrastructure mindset. The Golden Path requires "Day 100+" infrastructure mindset, thinking about the infrastructure as delivering value not just on Day 1 of project execution, but covering the entire sofware development process lifecycle. Montage provides concurrent governance and orchestration of packaged capabilities through highly extensible infrastructure that is adaptable to arbitrary workflows.

There are more concepts to be introduced related to Montage, but with these basics we can begin to describe the Virtuoso infrastructure.

Composable Abstractions

Virtuoso represents composable no-code platform engineering infrastructure, designed to build next-generation Golden Path workflows through composable, fully orchestrated software abstractions. This begins with the Virtuoso Core Framework and the schematic editor. The schematic editor provided by the Core Framework represents a single high-level environment where capabilities of arbitrary complexity can be effortlessly downloaded, installed, dragged, dropped, configured, and connected. Like any new programming language, the concept must be experienced through direct use for the full understanding and appreciation.

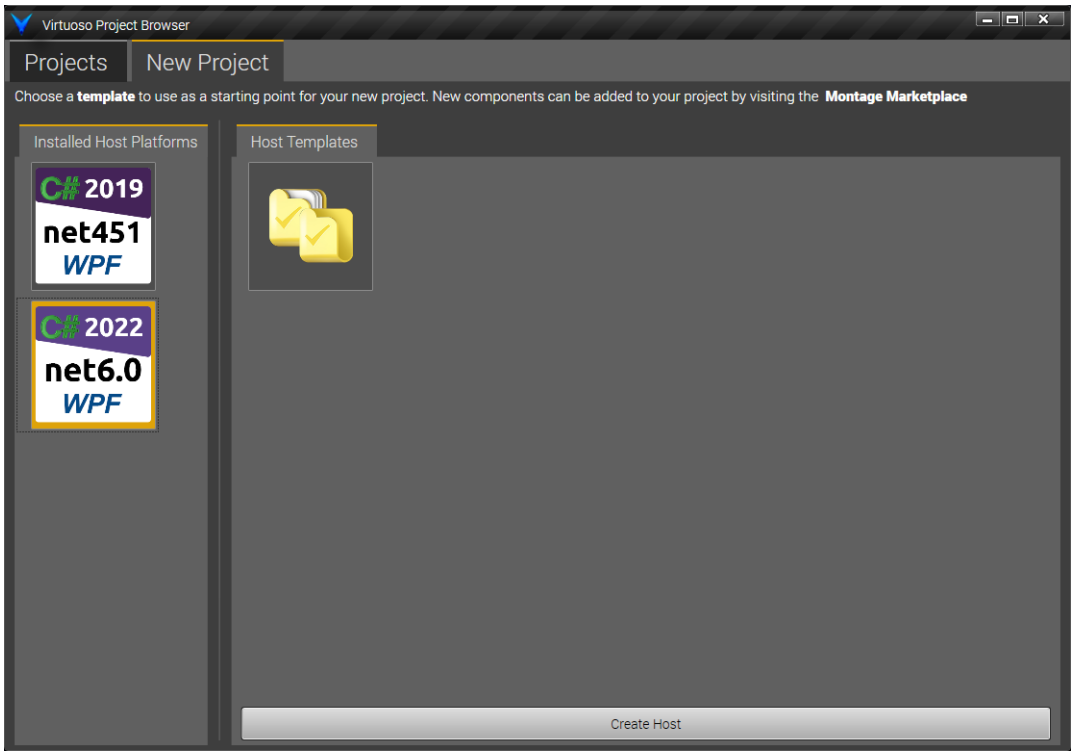
A foundational requirement of Virtuoso's infrastructure is to provide no-code abstractions on top of existing pro-code workflows. That is, the output that Virtuoso produces from its high-level visual programming environment is a fully-formed pro-code application ready to be built and run. You can use Virtuoso to design an application, then completely uninstall Virtuoso, and the application should build and run exactly the same as a traditional pro-code workflow. This satisfies key characteristics of the Golden Path not satisfied by conventiional no-code platforms. By flowing down to traditional pro-code toolsets and build systems, there is no lock-in or architectural limitations. The overall workflow is augmented with the power of no-node abstractions without any of the limitations.

The second requirement is that Virtuoso is architecturally designed to be extensible to any pro-code pipeline. The relationships between the high-level abstractions and the specifics of the pro-code programming language and build tools are standardized and extensible, so that anyone can extend the Core Framework to create no-code workflows for other pro-code pipelines or even non-coding related outputs from the visual design environment.

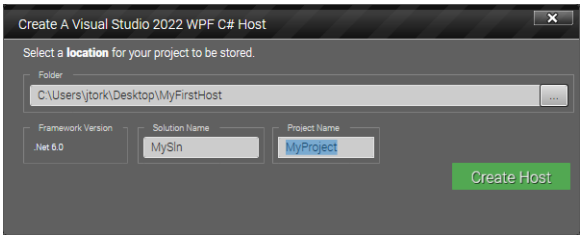
The third requirement is that "no-code" capabilities in Virtuoso represent high-level fully orchestrated capability abstractions that require absolutely no understanding of how they work, or what is needed to get the capability working. This requirement means extremely high levels of automation and orchestration. A no-code component or port represents an abstraction of a capability which may require software applications, SDKs, or drivers to be installed to the computer. No-code users should not be expected to hunt down packages and install them. All

orchestration and setup of the capabilities, including any tools needed to build them, must be fully orchestrated. On-premise orchestration is fundamentally a much harder challenge then cloud orchestration. Montage natively provides on-premises orchestration.

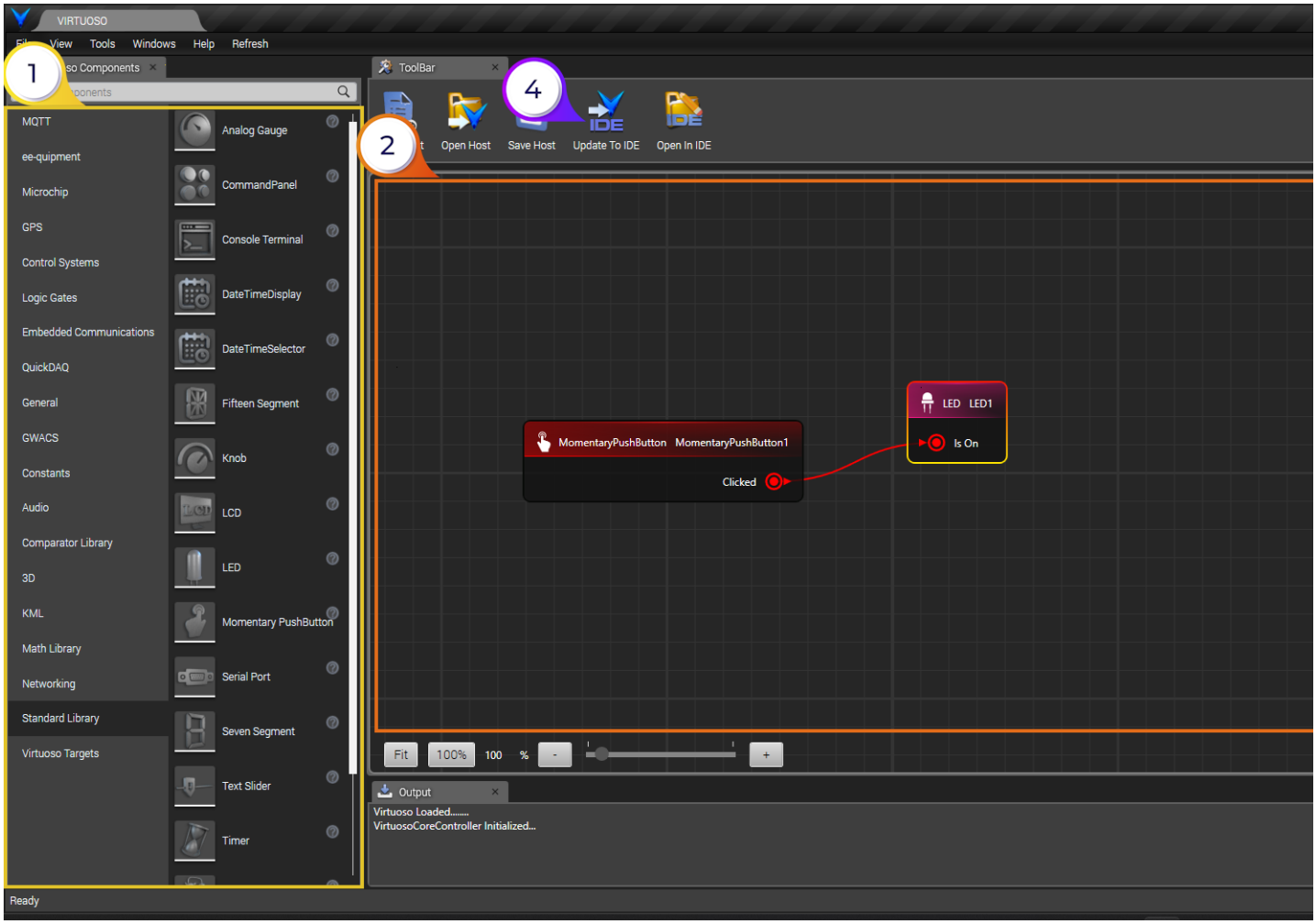
We can now review the major elements of Virtuoso, step by step. A user wants to build an application. First, they select a supported host platform,which is a specific main application type that the no-code visual programming environment represents, such as C#, C++, Python, Ruby On Rails, etc. In the image below, the user first selects a host platform from the available installed host platforms. Each host platform is itself a modular, discoverable Montage package. Once the host platform is selected, they can select a host template that someone has created for that host plugin. Host templates are also separate modular, discoverable Montage packages that anyone in the ecosystem can create and share (and sell!).



This "New Project" interaction is a part of the Virtuoso Core Framework, as we are in the process of selecting one of an unlimited number of host types provided by the Virtuoso ecosystem. Once the host and host template are selected (in this case a C# Windows Presentation Foundation application in Visual Studio), a popup window allows you to select an empty folder to create the new project in, select the .NET framework version as appropriate, and specify a solution name and project name. This popup interaction is an extension of the Virtuoso Core Framework, not part of the Core Framework, providing the specific interaction needed to complete project creation for the selected host plugin. This is an important architectural point for users wanting to extend the Virtuoso no-code environment to different uses and outputs. The selected project is then created from the template, renamed, and re-instantiated.



The host plugin then creates the host application, which is opened in the schematic editor provided by the Core Framework. The schematic editor represents the high level environment where a toolbox of no-code capabilities (1) can be dragged and dropped onto the schematic editor and connected to other capabilties (2). Each capability can be configured by a completely custom property editor designed specifically for that capability (3). This design is then pushed to the pro-code IDE or pipeline (4).



This specific "Hello, World!" example shows two very simple capabilities: a momentary pushbutton and an LED. The momentary pushbutton has a "Clicked" boolean output, and the LED has an "Is On" boolean input. Although this is a very simple application, it represents all elements of the differentiation process. The momentary pushbutton represents a differentiated capability with it's own development and release cycle. Once the capability has been fully developed, tested, and released, it is ready to be configured and composed with other capabilities, in this case an LED. In the case of the LED, after instantiating the LED there is a further differentiation step of selecting what color the LED should be and providing a display name. These two components are both provided by a Montage package with package ID of "Virtuoso.StandardLibrary". This package includes a dozen or more components, each with a custom property editor application. Other packages from other content creators can be installed to the **same environment**, instantiated, configured and connected to these capabilities, with each property editor for each component developed by different teams integrated into a single environment.

The momentary pushbutton and LED are about the simplest components imaginable, but components can represent arbitrary complexity. Many components do not require configuration, but some component property editors can provide extensive tooling to customize the specific behavior of the component. It is the responsibilty of the no-code component creator to provide the Golden Path for each individual component to maximize usability.

Scalability And Composability

Virtuoso is built entirely of modular and discoverable Montage packages, from the Core Framework, to host plugins, to host content. Thus the scalability of Virtuoso is multifaceted, as it scales to new pro-code host environments, like C#, Python, and Java, and within each of those host environments it scales with unlimited content. And it not only scales with with new content being added, but it scales with **composable** content. That is, content that can interact with other content with which neither content has no prior knowledge of. This is made possible through the separation of two clearly distinct Virtuoso package types: "port" packages and "component" packages.

Virtuoso port packages are Montage packages that define no-code port interfaces. In our prevous example, the momentary pushbutton and the LED both must somehow "agree" on how to share boolean state information: how one component can logically drive a boolean value, how another component can receive a boolean value, and how those two interfaces connect in pro-code. These two components could come from completely different packages, in which case this design would represent three different packages: a port package that defines boolean interfaces, a component package that provides a momentary pushbutton, and a component package that provides an LED.

In the above image, two port definitions are used: a booelan driver port and a boolean receiver port. Each port defintion provides a default visual style indicating how the port should look by default. When a component author instantiates a port on their component, the default visual style is applied to the port. This saves the component author time, and also helps provide consistency in the look of the ports. Thus, all boolean ports are red, and output boolean ports have an arrow that points outward while input boolean ports have an arrow that points inward.

However, component authors are free to override the default style and design port instances hosted by their components to look however the component author likes. In the example below, the virtual evaluation board has boolean input D7 and boolean output B1 located along the top of the board, close to the physical I/O on the board. In this case, default styles for both the component and its ports were overriden to create this. Virtuoso is designed to provide a state-of-the-art, powerful schematic editor that enables content creators to present their abstractions exactly like they want.

The Virtuoso.Port.DotNet.Standard package introduces the boolean interface as well as numerous other new interfacing abstract concepts to the Virtuoso ecosystem. Once a no-code port interface is defined for a Virtuoso host platform, the entire host platform ecosystem can make content that uses these port definitions. And as we will see, these interfaces can be very

sophisticated.

Port packages deliver content that is different from component packages. Port packages typically provide a single assembly that is installed to the host application that defines pro-code classes that are used to facilitate interoperability. There is no limitation on what can go into the port package assembly, and typically these assemblies contain substantial code bases. The assembly typically provides data classes containing the data that is exchanged between interfaces, as well as service interface defintions and service implementations. The specifics of what a port represents and how it is implemented is completely up to the developer's discretion. Port interfaces are easily designed by pro-coders familiar with the associated host environment. Port design is a low learning curve, but actually implementing a few ports is needed to fully understand the paradigm and the power that ports represent in scaling out ecosystems..

In addition to the pro-code classes each port provides in the installed assembly, port definitions also provide code snippets that are applied to components that expose port instances, as well as code snippets that are applied to the end application. This is an important point: port authors provide no-code design automation for two different future design stages: the component design stage, where the port is instantiated and used in the component, and the host design stage, where port instances instantiated by components are connected to other components. Component authors essentially benefit from no-code infrastructure provided by port authors, as port authors provide all the code to instantiate port instances, so that it's all done automatically for component authors. They simply instantiate ports from any port package, and their component is configured with the pro-code representation of the ports. From there, they write pro-code business logic to interact with the port as necessary. After the components are designed and deployed by component authors, the components are instantiated and connected to other ports. The connection between ports is done not through code emitted into the component, but into the host application itself.

A port of a specific type can connect to other ports if any port has defined code snippets to connect between them. Thus, ports defined inside a package can internally provide connection snippets, such as how to connect a boolean output to a boolean input. A port can also define connection snippes to other ports in completely different packages. In this way, the first port has prior knowledge of the other port, wheras the other port has no knowledge of the first port. In this way, the Virtuoso environment represents true no-code composability between content creators. As will be seen, this has important economic implications. When ports provide connection snippets for ports in other packages, a dependency is automatically formed on the other package.

Maintainability And Productivity

Before addressing network effects between organizations, a clear value stream realized from internal organizational use can be seen: maintanability.

The Virtuoso no-code environment represents an infrastructure-supported pre-differentiated environment where an application design can be broken down into high levels of abstraction with clearly defined interfaces between application elements. The declarative visual programming environment and Montage package based modularity enforces a composable architecture with clearly defined interfaces.

In a typical professional development team with multiple developers, devlopers may develop independent modules, but they all get lumped back together in the build process's jump from zygote to differentiated animal. The application design isn't itself maintained at a higher level of abstraction. Whereas for applications designed using Virtuoso, the entire application can be represented and visualized in the schematic editor. This makes comprehension of the application architecture and onboarding new resources significantly easier. Golden Path support for frictionless onboarding of new resources is a stated responsibility of traditional platform engineering, however traditional platform engineering does not cross the threshold of the integrated development environment or software implementation itself.

Maintainability includes maintenance of the team's tribal knowledge. Remembering the implementation logic can at best be supported by good documentation of code, but for complex software, even the original implementer may lose mental track of the original implementation and reimplement business logic, simply due to memory challenges. The Virtuoso no-code environment and declarative design pattern creates and maintains a persistent application structure with a level of abstraction that unburdens developers from keeping the entire system architecture in their head. This also is a stated responsibilty of platform engineering in the pursuit of the Golden Path.

Network Effects, Economic Integration, and the Nature of the Firm

Platform engineering is traditionally associated with the development of *internal* development platforms specific to the organization, however the Golden Path requires infrastructure that can support fully orchestrated packaged capabiltiies that are shared between organizations on a two-sided platform. Powerfully disruptive network effects are created from the Virtuoso no-code infrastructure with discoverable content that is fully orchestrated by Montage. To understand these network effects, and to leave nothing to the imagination, an economic deep dive is needed.

Software production may be considered a service economy that suffers from Baumol's cost disease, an economic disease that affects goods and services whose production are characteristically resistant to fundamental improvements in productivity. Like haircuts, medical care, or college tuition, certain services must be provided by skilled manual work. Characteristically, these production economies are resistant to productivity improvement through economic innovation, compared to other goods and service production economies like consumer electronics. Improvements in the means of production in other areas translates to higher standards of living, which translates to disposable income, which translates to more demand for goods or services affected by Baumol's cost disease. This "demand elasticity" results in even higher costs for services affected by Baumol's cost disease, like software production.

The advent of large language models has been perhaps the only meaningful technological innovation that can address Baumol's cost disease for software production. While there is still generally a low opinion that generative AI will *replace* software developers, improvements in software developer productivity of up to 20% (for those organizations that allow its use) is commonly cited. Some opinions hold that generative AI's productivity gains can double developer productivity, but even this is not a truly disruptive economic shift.

Nonetheless, Baumol's cost disease, as it relates to software production, can be alleviated. The insights provided here are primarily those of economist Ronald Coase, boiled down and mapped to software production. Readers of this introduction are encouraged to read his article "The Nature of the Firm", to fill in gaps that can't be filled in this intorduction. References to "the firm", the "individual", "entrepreneur", "in the firm", "economic coordination", and others may otherwise be misunderstood. Coase's interests lay in understanding why firms exist, and what determines their sizes. He determines that transaction costs are the key drivers to the optimization of forms of economic coordination that occur as either actions of individuals or actions of a firm. An important theoretical distinction is necessary to distinguish the firm and the individual. The "individual" coordinates production through market prices alone. The classic example is a quality shirt made in Malaysia using German machines from cloth woven in India using cotton grown in the United States. This chain of production is organized by market prices at each step of the supply chain. The supply chain participants of a quality shirt market then act in concert, performing the unconscious calculus that compares costs, features, and qualities, to select suppliers that have found the optimum chain of production.

While each supplier also obviously contributes to coordination, the primary coordination of this production is understood to be the unconscious market computation of prices as the aggregation of supply and demand. From these market calculations, individuals act. This process is akin to the power of quantum computing to arrive at the best solution from a nearly unlimited set of possibilities. Coase's insight was that the production coordination of firms is different from individuals coordinating production through market prices. A firm sets out to produce economic value, certainly affected by market prices of inputs, however coordination of production here is something different. What is referred to as the firm's "entrepreneur" or "undertaker" does not know exactly how much it will cost to, for example, take old rockets, add autonomous vertical take-off and landing, and completely revolutionize the means of producing space launch capability. This involves a conscious agency of a firm and its entrepreneur to coordinate the specialization of labor to produce something that market prices cannot.

The entire gestalt of economic production is thus described in "The Nature of the Firm" as a sea of unconscious market computation, with the entrepreneur's coordination representing an "*island of conscious power*". The firm entrepreneur "*busies himself with the division of labour inside each firm and he plans and organizes consciously*," but "*he is related to the much larger economic specialisation, of which he himself is merely one specialised unit. Here, he plays his part as a single cell in a larger organism, mainly unconscious of the wider role he fills. [sic]*"

Coase's Theorem includes interesting insights into factors of production, transaction costs, and how they shift the equilibrium of production coordination between individuals working with market prices, firms making conscious internal coordination, and firms coordinating with other firms. Each type of coordination involves transaction costs, and while unpacking them further is interesting, here only two improving shifts in production will be presented. In a strictly economic sense, a "combination" occurs "*when transactions which were previously organised by two or more entrepreneurs become organised by one. [sic]*" All things being equal, this is an improvement in production, as it only requires the transaction cost of one party. Beyond "combination" is an economic "integration", which occurs "*when it involves the organisation of transactions which were previously carried out between the entrepreneurs on a market. [sic]*".

"The Nature of the Firm" provides economic theory that makes software production more understandable and can avoid costly architectural decisions. The theory can also provide proof of value. We attempt to do both, first by reviewing the function of a software platform. Some platforms represent a "combination" in the economic coordination sense of Coase's Theorem. Before the platforms were built, multiple entrepreneurs had to transact together to produce a capability. Once the platform is created, it increases software production by allowing users to build out capabilities leveraging the platform. The transaction costs involved are then incurred by only one entrepreneur, the user that uses the platform to produce a specific use case solution.

Other software platforms increase software production by replacing coordination (software development) that was previously performed "in the firm" and instead moving it outside the firm. Modern no-code and low-code platforms are an example of this. Here, the applications are developed by platform users that leverage no-code tooling to create capabilities. Some economic coordination (software development) is performed by the platform user, however most of the actual economic production is done by the no-code platform. This is an example of

a no-code platform creator entrepreneur coordinating production to create a means of production mediated by market action: No-code users select a platform based on price and quality, sharing the cost of production. This technology, when applied successfully, results in a natural reduction in the size of the firm, as the transaction costs of using the platform are much lower than the internal transaction costs of coordinating the same production without the use of the platform.

In either example described, the combining coordination of platforms or the outsourcing coordination of no-code platforms, we see three missing processes of economic production described earlier. **First**, there is no economic integration. That is, there is no technology that supports a means of software production which previously required at least one entrepreneur, and now requires no entrepreneur coordination. Again, by "coordination" we refer to where actual economic production occurs. This missing first process results in the **second** missing process, which is a hierarchical orchestration of the means of production. Again citing Coase, "*[the firm's entrepreneur] is related to the much larger economic specialisation, of which he himself is merely one specialized unit. Here, he plays his part as a single cell in a larger organism, mainly unconscious of the wider role he fills. [sic]*". In the wider economy, means of production form "holons", which are simultaneously a whole in and itself as well as a part of a larger whole. This process involves firms, in their conscious economic coordination, creating a higher-order emergence of coordination that is again unconscious. This second missing process results in the third missing process, which is a functioning marketplace of higher order composite capabilities, coordinated unconsciously by price, yielding enormous productive power to the firm's entrepreneur.

The goal of this introduction section is to firmly establish these economic understanding in the minds of software infrastructure architects. The understanding clarifies the economic value that some platforms have, while also clarifying where their value might currently be limited due to a lack of infrastructure. We can also use this understanding to predict that modern no-code and low-code platforms will likely be extinct in 5 to at most 10 years. These business models require a platform service value stream, where they are compensated for providing capabilities that the firm internally coordinates/produces. These modern no-code and low-code platforms have nonetheless demonstrated their value. Competition among these firms is fierce, due to the enormous market size and the overall lack of key differentiators. And because all the productive coordination of capabilities must be done inside the respective no/low-code platform provider firms, the amount of capital that these firms require, and have attracted, is eye-popping. In our analysis, these investments in software production, where they can't be properly re-architected to facilitate the three additional spheres of economic software production, will prove to be evolutionary dead ends. To understand why, we must consider infrastructure that enables these additional spheres of software production coordination.

Montage Package Mechanics, Economic Integration, and Opinions

The previous section described the importance economic integration, and the necessity of removing transaction costs between different economic actors to accomplish this.

Both Virtuoso and Montage are so-called "opinionated infrastructure", and this is a good thing. The layering of Virtuoso infrastructure on top of Montage infrastructure is itself telling of the opinionated design. And whether you agree with the opinons or not, it is very helpful to clearly understand these opinions.

Montage orchestration is organized around several different concepts:

Montage Activity: A Montage "Activity" is a fully orchestrated design project or activity that a user wishes to work on, on their computer (on-premises), or elsewhere. The Activity represents the project or projects being worked on, as well as all software, SDKs, drivers, and everything else needed to build and run the project. For clarification regarding the difference between Montage Activities and build system generators, see here. Montage Activities represent the entire dependency graph of packages necessary to guarantee that once the packages are unified and installed, the activitiy will build and run.

Packages: Packages are the modular blocks that are orchestrated by Montage Activity dependency graphs. Packages can be Montage packages or native packages used in specific ecosystems. Montage infrastructure is designed to flexibly adapt to existing package management systems to unify the orchestration of packages across traditional ecosystem barriers. Both Montage Activities and Packages are purely abstract concepts. What they actually represent gets solidified when Montage is adopted to specific ecosystems.

Installation Targets: Montage orchestration involves, among other things, logically "installing" packages into 'things" that packages can be installed into, which we generically call "installation targets". A C# project will typically have Nuget packages installed into it. In that case the Nuget package corresponds naturally to the Montage generic notion of a "package", and the C# project itself is the installation target that the packages are installed into.

Dependency Graph Unification: Packages commonly have dependencies, and before a package can be installed into its installation target, all package versions must be "unified" to ensure that all dependency requirements are satisfied. If a dependency can't be satisfied because it conflicts with another dependency, then the package can't be installed.
Unification Domain: Montage Activities represent packages that need to be installed into installation targets. The set of all installation targets that the Montage Launcher must orchestrate is called the "Unification Domain". Unification domains are currently limited to a specific PC, but the scope of unification domains and Montage's orchestration will grow.

Target Packages: During your use of Virtuoso and the Montage Launcher, you will see references to "Target Packages". A "Target Package" refers to a particular package being installed into a particular installation target. There may be five different C# projects on my PC. The single package "Virtuoso.StandardLibrary" may or may not be installed to each installation target. The Virtuoso.StandardLibrary package could unify to version 1.0.0 in installation target "A", but unify to version 2.0.0 in installation target "B". Because the same package is installed into different installation target, they unify independently, and the combined package and installation target scope is called a "Target Package".

Heterogeneous Unification: Traditionally, package management systems support specific packages for specific native projects, like C# (Nuget) or Python (PIP). These "homogeneous" package management systems support orchestration of packages within a single installation target, all of the same type. Montage supports orchestration of packages with dependencies that can cross boundaries between installation targets and package types. This is where substantial power is created.
Governance: Package management systems also typically do not provide extensive governance in the form of policy or licensing enforcement. Montage provides this as a core part of its orchestration service.

The full picture of Montage's power and flexibility involves Golden Path infrastructure that supports the content consumer side *and* the content producer side of a two-sided platform. The Golden Path for consumers involves effortless ease of use of content, whereas the Golden Path for consumers includes all infrastructure related to managing and even monetizing content. With these basic concepts in place, we can walk through a step-by-step example.

Appendix

.

Spiral Dynamics

Spiral Dynamics is a theory that describes the evolution of the intelligences that are adapted at the individual and societal level. It describes distinct stages of growth that both individuals

and societies go through. Characteristically, each stage of growth solves problems that were left unresolved from the previous stage, while in the process creating new, typically more complex problems. A common characteristic of each stage of growth and formulation of new "intelligence" is a belief that the new stage is superior to and should replace the previous stage. In contrast, more developed so-called "integral" stages of development recognize that each stage of development represents intelligences adapted for a specific environment or "life condition". The integral perspective views each stage with validity, and seeks only healthy expressions of whatever intelligences have been adopted based on life conditions, as opposed to unhealthy expressions that commonly associate with each stage.

The integral perspective provides a flexible conceptualization of different technologies. It offers a process-oriented perspective where newly adopted intelligences should transcend but include previous intelligences, as opposed to attacking or replacing them.

In the contex of technology, the integral prespective as it relates to Virtuoso recognizes that software development technologies represent holons of hierarchical complexity. Pro-code technologies, programming languages, and frameworks, in their variety and sophistication, are suitably adapted for the variety of use cases for which they were developed. They are whole in and of themselves, yet from the high level of abstraction that Virtuoso infrastructure creates, another emergent unique level of software technology produces distinct composable parts that can be used to create whole software applications.