uniform

Demystifying MACH Architecture in a Composable World

Uniform whitepaper

Q2 2021

Demystifying MACH architecture in a composable world

Making enterprise technology decisions can be difficult, confusing, and stressful, especially when they're decisions that you and your company may have to live with for years. The technology buying process practically requires accepting the risk of being locked into products that are not able to keep up with your organization’s changing needs, or even worse, products that were never really a good fit in the first place. So, it is no surprise when an architecture comes along that promises to reduce that risk, people tend to notice.

One such architecture that is growing in prominence and popularity is MACH. At its core, MACH is a decoupled architecture made up of modular, self-contained, and completely independent components that work together as one. In fact, MACH can be seen as a direct response to the monolithic architectures that have dominated enterprise applications until quite recently. It strives to address the limitations of the previous generation of technology and to take advantage of the capabilities of modern technology while attempting to remain flexible enough to adapt to new changes as they come.

Because it's a relatively new architecture, MACH is still shrouded in a certain amount of misunderstanding and mystery. To better understand MACH, we need a better understanding of two concepts that are foundational to MACH architecture: integration and composability. Likewise, to fully appreciate the value of a decoupled architecture like MACH, it is important to examine the advantages and disadvantages of its monolithic predecessor.

What does MACH mean?

MACH is an acronym that stands for microservices, API-first, cloud-based, and headless. Each of these qualities describes a specific, technical hallmark of a MACH architecture. Together, these qualities promise to reduce the risk of product lock-in and increase the ability for organizations to adopt the technology that is best able to meet their needs, when they need it.

Monoliths & menus

The chief alternative to a decoupled architecture like MACH is a monolithic one, a system that provides all of its functionality in one big interdependent package. A monolithic architecture is sometimes described as being “tightly coupled.” Some of the differences between these two distinct approaches to building enterprise applications can be illustrated with a simple analogy.

It's dinnertime and you're hungry. You've got a number of options. You can go out to a restaurant, or you can stay home and cook for yourself. Eating dinner at a restaurant has one big advantage. It requires no work on your part. On the other hand, it costs more, and you're limited to what's on the menu. This option can be fine if you're a bad cook or if you go to a good restaurant. But it's also a bit of a gamble. If you choose a new restaurant, you may not like your meal. And even if you stick with your favorite place, the chef may have changed, or your favorite entree may not be available.

Another option is to stay at home and cook for yourself. Of course, this will require that you do all the work. On the other hand, it will cost less, you'll get what you want, and you will be able to make it exactly the way you want it. For example, if you want to put cheese on your fish or sprinkle chocolate on your spaghetti, you can do that.

There's nothing inherently better about eating out at a restaurant or staying home and cooking for yourself. Which option is better depends completely on context.

A monolithic architecture is a bit like eating at a restaurant, while a decoupled architecture is like a home-cooked meal. Of course, choosing between a home-cooked meal and a restaurant can be very subjective and depend on a lot of other variables. (Is it a weekday or a weekend? Are you tired and content to stay home or antsy and anxious to get out?) So, it's not a perfect analogy to choosing a decoupled or monolithic architecture.

In fact, choosing between a decoupled and monolithic approach is much easier than deciding whether to go out or to eat at home. You can usually make your decision by answering just two basic questions: Are the advantages greater than the disadvantages? And are the advantages even real?

The challenges of maintaining a monolith

Let's take a closer look at some of the challenges involved with using a monolithic system. The main problems with a monolith are its complexity and interdependencies. A change you make in one place can result in failures downstream.

That's why regular changes, such as system updates or upgrades require a huge amount of testing in order to make sure that they don't wind up breaking something somewhere in the system. An upgrade can literally take months to complete. And those are months during which no business value is coming out of the monolith. The same applies when it comes to adding new features. If you've ever wondered why your developers are telling you it'll take a month just to add a page to your Web site, that's why.

You might find yourself asking: is this really a problem with the monolith? Or is it just poor design? Could a monolith be designed in a way that avoids these complications?

In theory, the answer is yes. A well-built monolith could be set up to avoid most of these problems, although the truth is that no one ever deliberately set out to build a monolith. It wasn't designed. It evolved. Monoliths are less a product of planning and more the result of mission creep. In this respect, monoliths aren't truly sustainable: that is, they can't easily adapt to changing needs – even though the ability to adapt to changing needs is critical.

What is a suite?

A monolith is sometimes confused with a suite. And vice versa. A suite is a product made up of a variety of other products maintained by a single vendor.

One of the best-known suites is Microsoft Office. Some products in the suite may have been built by the vendor, while other products may have been acquired elsewhere. In the case of Office, Word and Excel were developed in house, while the original PowerPoint was acquired from another software company.

What about Jamstack?

Like MACH, Jamstack is a decoupled architecture. The difference is that Jamstack is limited to the Web channel. MACH, on the other hand, is an architecture for enterprise applications. Although it can be applied to the Web channel, MACH can also be applied to any other digital channel.

Jamstack describes how a Web application should be designed and has some opinions. It specifies some but not all of the technology you should use, most notably JavaScript, which accounts for the J in the Jam. This makes sense when you consider that JavaScript is the de facto standard for Web technology. Contrast this with MACH, which is explicitly agnostic when it comes to technology.

MACH specifies how a platform should be designed but not what you should use in order to implement it. Jamstack and MACH appeal to people for similar reasons: flexibility, modularity, modern tooling, efficiency, etc.

Since Jamstack is the older of the two technologies, it can be helpful to look at MACH from the perspective of Jamstack. If you thought about the problems that Jamstack solves and wanted to design an architecture that solved the same sorts of problems not just for Web sites but for any enterprise application, you'd probably come up with something very similar to MACH.

Although the terms suite and monolith are often used interchangeably, they are different things. A suite is an actual product you can buy, while a monolith is an architecture. It describes the way that a product has been designed and built. Suites often use a monolithic architecture, but not always.

Even though a suite may not be built on a monolithic architecture, most of them are. This can make upgrading and maintenance a challenge. Customizing the suite to accommodate your company's special requirements further increases the degree of difficulty and introduces even more potential pitfalls when upgrades and maintenance are necessary.

Another disadvantage of a suite is the implementation cost. Sometimes the cost of service for getting a suite implemented is actually greater than the cost of the suite itself. This is because of the customizations mentioned previously. It is exceedingly rare that a suite meets an organization's needs out of the box. Extensive – and expensive – custom development projects are required in order to get the suite to work the way an organization wants.

Things get even worse as a suite gets older. Once it's labeled a legacy platform, developers begin to move away from it. As their numbers dwindle, the developers that remain become more expensive. The most experienced and skilled developers are usually the first to move on to new challenges, creating a "brain drain." Customers end up literally paying more for less.

Suites vs. MACH

How does a suite compare to MACH? The difference really comes down to choice. With a suite, it's the vendor who selects the products for you. With MACH, you select the products you want from the vendors you want to buy from.

Back when suites were gaining popularity, building a stack yourself was just not practical for most organizations. These days vendors are building their products with the expectation that companies will be integrating them with other products. Not only that, but the technology infrastructure available to deliver these products from the cloud means that organizations don't need to worry about how they'll support multiple products built with different technologies. This is the world that MACH enables.

For an organization that wants to build a technology stack made up of products that are right for its unique requirements, MACH provides the technical foundation. But just as a monolith isn't a product, neither is MACH. It's an architecture for building products. What this means is if you buy a CMS or a merchandising tool or a personalization tool or any other enterprise module that's built on MACH principles, you should be able to connect them in a clear and consistent way.

New technology, new options

The transformation of retail provides a good analogy for this change. Consider a typical shopping mall. It offers most of what you need and provides some consistency during your shopping experience. However, malls don't sell everything, which means you have a limited selection from which to buy. Using a suite is like shopping at a mall.

Although online shopping was limited at first, once the proper infrastructure was in place, it quickly replaced shopping malls as the preferred shopping destination. Granted, there are still some cases where an in-person experience is preferable, but the list of shopping that you can't do online is getting shorter and shorter every day. The experience of shopping online is like using MACH architecture.

What is composability?

When talking about technology today, the term composability is probably used most often to describe composable digital experience platforms (DXPs) and composable commerce. Gartner describes a DXP as "an integrated set of core technologies that support the composition, management, delivery and optimization of contextualized digital experiences."

Composability enables you to select the best components, and you get to determine what "best" means. Composability doesn't describe the quality of the products or where they came from; it just describes the ability to select the products that you choose to connect.

Who chooses the components?

Once again, it comes down to choice. Who selects the components that make up the DXP? What makes a DXP composable is that you are able to select the products that make it up. In fact, you don't simply have the option to select your components; you have to. Luckily, the components available to choose from are designed in a way that they can be connected with one another.

Contrast this with a suite, where the DXP comes with a set of pre-integrated components that are already selected. Even if its components can be bought as stand-alone products and can be used in a composable system, the suite is not composable. That's because you cannot choose the components yourself.

Let's be clear: Suites aren't a bad thing. In fact, for certain requirements, a suite may still be the right option. Yet, there's a reason why composable systems are on the rise. A composable approach offers more benefits and fewer disadvantages. Suites were once the only option. That was when integration wasn't worth the cost. But integration has steadily gotten easier and more affordable, and composable systems are growing as a result.

What is integration?

Integration is relevant to every tech stack, regardless of whether it's monolithic or decoupled, or whether it's a pre-integrated suite or a composable system. At a very basic level, integration is how separate systems or products communicate or work together.

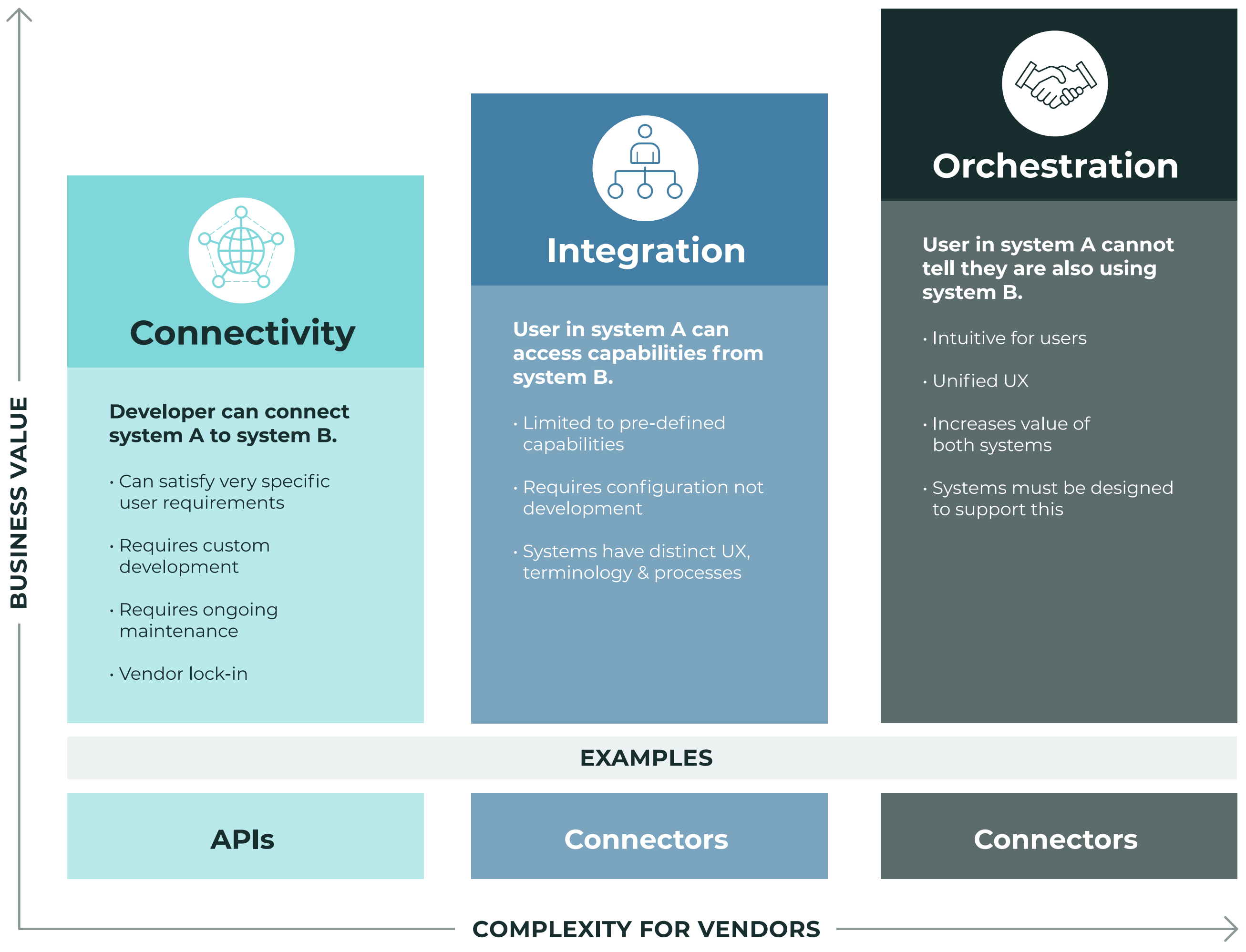
Beyond that, the definition of “integrated” can vary. When someone says a system is integrated, it can be difficult to determine exactly what they mean. Rather than simply saying, ”This is integrated” and “This is not integrated”, it is important to ask ”How well is this integrated?” or even just “How is this integrated?”

The integration maturity model

At Uniform, we've developed a way to categorize the available kinds of integrations that we call the integration maturity model. The integration maturity model is designed for technology buyers and decision makers, business users, technical architects, and developers. It shows them what to look for to determine how well the systems are integrated, how to recognize the signs of an immature integration, and how to determine if an integration is mature enough to be useful.

The model defines three types of integration. The first is connectivity, where a connection can be made from one system to another. The second type is just called integration, and it's where a user can access one system while working within another. The third type of integration is orchestration. This is where the integration is so seamless that users operating within one system can't even tell that they are also making use of another system, enabling them to focus more on their work rather than on their tools.

Uniform Integration Maturity Model


Connectivity
Developer can connect system A to system B.
· Can satisfy very specific user requirements
· Requires custom development
· Requires ongoing maintenance
· Vendor lock-in
Integration
User in system A can access capabilities from system B.
· Limited to pre-defined capabilities
· Requires configuration not development
· Systems have distinct UX, terminology & processes
Orchestration
User in system A cannot tell they are also using system B.
· Intuitive for users
· Unified UX
· Increases value of both systems
· Systems must be designed to support this
EXAMPLES
APIs
Connectors
Connectors
BUSINESS VALUE
COMPLEXITY FOR VENDORS

Integration in depth

What do each of these levels of integration – connectivity, integration, and orchestration – involve? What do they look like?

Connectivity. APIs provide the main example of connectivity, the most basic form of integration. APIs can satisfy very specific user requirements. They give you direct access to the system. But because there's no user interface with an API, some level of custom development is always required. And, as with any custom development, ongoing maintenance is necessary.

One of the big drawbacks in working with APIs is that it creates vendor lock-in. Your components are limited to communicating in a way that your vendor has defined. This is particularly problematic when a system doesn't meet all of your organization's requirements in the first place. You wind up customizing things in a way that locks you into the vendor even more.

Integration. Integration usually comes in the form of connectors. These are pre-built components that connect specific systems in a specific way. The purpose of a connector is to try to create a bridge between systems that have distinct processes, terminologies, and user experiences. Of course, these connectors only work if your system is supported. Otherwise, they're useless. Because connectors are pre-built rather than custom made, they are limited to certain predefined capabilities. Rather than requiring the sort of custom development you need to do with APIs, integration requires configuration.

Orchestration. Orchestration relies on extensions. Extensions take advantage of features written into a system that allow developers to add new functionality. The best extensions are intuitive for users. They take a known and familiar system and add a new feature to it. Because an extension literally just extends a system, it can take advantage of that system's existing user interface, increasing the value of both systems by making both easier to use.

Of course, extensions don't work with all systems. A system must be designed to accommodate them. Unfortunately, some systems that support extensions may not support the kind you need. If so, you might have to build a connector or use the API.

In order to make the differences between connectivity, integration, and orchestration a little clearer, here's a concrete example. Suppose you have a CMS and you want to use images from a DAM (digital asset manager) on your site. With connectivity, you can get images from a DAM and use them with a CMS, but you'll need a developer to set this up by writing to an API. With integration, the vendor (either the CMS vendor or the DAM vendor) has built a connector that allows you to import images from the DAM.

With orchestration, an extension means the CMS is capable of getting images directly from the DAM, on demand. The communication between the two components is so seamless that it's virtually unnoticeable.

Everyday integration

In general, integration describes two things working together in some way. It's not a concept that is limited to technology.

If you've got air conditioning at home, for example, you could say that your home is integrated with your air conditioner. How does this map to the integration maturity model? For connectivity your house needs to have electrical outlets, you need to buy an air conditioner, and you need to plug it in. For the integration (the second type), you need central air conditioning, and your electrical system must be able to power the appliance. With orchestration type integration, you live in a smart home that turns on the A/C whenever you're at home and the room temperature gets above 77 °F (25 °C).

See the difference? With orchestration, unlike with connectivity or integration, your attention shifts away from operating the air conditioner and focuses instead on the activities you're engaged in that require a comfortable temperature.

Who's going to build this thing?

Our efforts to demystify concepts like composability, integration, orchestration, connectivity, and decoupled architecture have left one crucial question unanswered: Who actually builds this? That is, who is going to bring all the pieces together?

Widespread adoption of any technology requires that some sort of infrastructure be in place. Telephones were a novelty until phone lines connected residences. Tesla is building an infrastructure of charging stations in order to drive adoption of its electric vehicles. In order for the Web to work, people needed Internet connections and had to have browsers on their machines. And, of course, the Web required content as well as tools to create that content. E-commerce had similar infrastructure needs. Among other things, there had to be ways to support payments and shipping. The same goes for SaaS (software as a service). After all, the people who build SaaS products don't build their own clouds. In addition, support for security, licensing, and scaling are part of the infrastructure that made SaaS possible.

People who want to build composable systems depend on infrastructure too. Although orchestration supplies the most business value, it's also the most complex to implement. While the MACH architecture and modern technology make it easier than ever to integrate, orchestration is still a challenge.

And everyone needs orchestration: it's infrastructure. As with any other technology, the infrastructure must be in place before that technology can be adopted on a large scale. Composable systems will be widely adopted when orchestration is readily available.

Uniform provides the infrastructure for composable systems

Uniform provides you with a fast track to composability by handling all the difficult and time-consuming tasks involved with integration. Although the infrastructure described above is absolutely necessary, building it doesn't provide any business value to you. The stuff you want to focus on are things like Web design and content creation, not the mechanics of how the settings in your personalization system are exposed to content creators working in your CMS.

Luckily, Uniform offers composability right out of the box. You select the components for your stack and we handle the connections. Uniform also handles orchestration across your stack. You don't have to build this complex but very valuable integration yourself. We already have.

A composable system built on a MACH architecture provides an example of a modern sustainable approach that addresses a lot of the difficult problems that organizations have accepted as a natural part of working with enterprise software. Composability allows you to build stacks out of tools you choose. MACH gives you an architecture to make that happen in a sustainable, scalable way, and Uniform provides the infrastructure that enables orchestration across your MACH platforms.