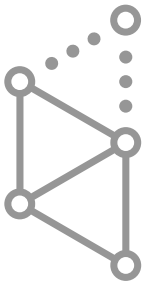

SOCIETAL PLATFORM

Societal Platform Design

Note on Design and Technology Characteristics

CONTENTS

Introduction

A Societal Platform approach to resolving a societal challenge necessitates the use of technology to achieve scale, speed, and sustainability. Though the specific technology features developed and deployed will depend on the problem being resolved; the approach to determining and defining them will be rooted in certain key design and technology characteristics. These characteristics in turn realise the core values of a Societal Platform.

This document presents these characteristics, along with examples; and is intended for social entrepreneurs and tech architects looking to apply the Societal Platform approach to resolving a societal challenge.

Societal Platform actors may be classified as Builders, Extenders or Participants. That is, those who contribute to the buildout of the platform itself, ie. build or extend the context-invariant elements, and those who consume a Societal Platform, ie. build context-specific solutions or use them. Those looking to build or extend a Societal Platform should internalise the characteristics presented herewith and understand how they reflect and realise the core values of a Societal Platform. These should motivate the many design and technology choices that will need to be made at various stages of the mission, across buildout, adoption or assimilation of the platform. The builders and extenders will need to balance these overarching characteristics with the demands of the mission at hand, as well as the available resources.

Societal Platforms and Technology

The success and stickiness of a Societal Platform lies in its ability to unlock scarce resources, create an environment for co-creation, and to catalyse solutions for diverse communities. The overarching goal being to reach and resolve for every single person affected by the problem. While this goal may be realised by one or more missions and platforms, technology infrastructure becomes crucial to all of them, for it affords them the scale and speed of the digital world.

Viewed as an inverted funnel, a base of context-invariant technology elements (infrastructure, if you will) must enable breadth of use while the context-intensive technology elements (solutions, if you will) must enable depth of use. While the infrastructure is focused on making a Societal Platform usable by all, the solutions are focused on making it useful for a certain someone. A co-creation process across all ecosystem players bridges these two seemingly contradictory objectives. The architects for their part would make the calls on how to spread the technology features across this spectrum. In other words, determine which technology element belong where and why.

Societal challenges are often rooted in cultural practices and mindsets that hold a power of their own. To assume that technology alone can resolve these will be foolhardy. Thus the underlying technology should be judged not on how well it solves a specific problem, but on how effectively it enables and empowers people to come together and solve for their contexts. In addition, to assume that all possible uses and implications of technology can be predetermined will also be foolhardy. Thus the underlying technology should be judged on how evolvable and extensible it is. To achieve these objectives the architects of a Societal Platform should adopt a 'whole-system design' approach. They should think about (note: not building for) the unknown future possibilities and balance that with the needs of a specific mission and a specific context. The quantum of importance they wish to assign to which aspiration is a choice they should own, justify and reflect in their design choices.

The success and stickiness of a Societal Platform lies in its ability to unlock scarce resources, create an environment for co-creation, and to catalyse solutions for diverse communities.

Existential Questions

The ‘whole-system design’ approach is a fundamental mindset for an architect. This mindset may be broken down into few existential questions that the architect constantly ponders over. The architect needs to keep evaluating how the design and technology choices, especially for the context-invariant elements, affect these objectives and outcomes.

The questions are:

1. How may we engage, enable and empower the ecosystem; to consume + contribute?
2. How may we enable trust on the platform, and also for the platform?
3. How may we leave/provide maximum possible agency with/to all participants?
4. How may we catalyse the network to resolve for all, resolve at population scale?
5. How may we enable participants to continuously evolve (the solutions) through data?

The ‘whole-system design’ approach is a fundamental mindset for an architect.

Design Characteristics

The dual objectives of ‘enables and empowers people to come together and solve for their contexts’ and ‘evolvability and extensibility’ along with the mindset of ‘whole-system design’ and the existential questions to help achieve those, all translate to a set of characteristics and patterns that the technology elements reflect. The following sections discuss these in detail. Again, the importance an architect gives to any characteristic is something they own, justify and reflect in their design choices; enabling future possibilities, while meeting mission objectives under present constraints.

Designing for Evolvability

Atomic

Designing the software elements as single purpose atomic elements that can then be recombined and layered to execute specific tasks ensures wider scope of use for the individual atomic elements and evolvability of the overall platform. Some simple checks for atomic elements would be to:

1. Deliver one and only one purpose
2. Purpose is less, but no less: i.e., offered value is high
3. Limit access via well defined (and limited) set of APIs[1]
4. Be removable and replaceable independent of other elements
5. Be reusable across applications/solutions, contexts, domains and even platforms

For example, an element that does encryption and decryption of data can be designed as an atomic element. It can be designed as a black-box with simple single purposes APIs that take data in and return encrypted or decrypted versions of it. The box itself serves just one purpose of encryption and decryption, and remains ignorant of the nature and format of data, the purpose of encryption/decryption, the entity requesting its services etc. This way, the same atomic element is useful to multiple applications, across contexts and platforms. Another example, a financial inclusion platform may use it to encrypt financial transactions on its platform, a healthcare registry may use it to encrypt patient records and so on. As the key management algorithms improve, the black-box can be enhanced or replaced without breaking services for any entity. On the contrary, the same element could also be non-atomic in design by customising operations for different callers, by exposing its key management algorithms and allowing entities to extend or customise them or by limiting services to data in specific formats only.

[1]Application Programming Interfaces

Extensible

To permit evolution as well as be useful under diverse contexts, missions and platforms, the technology elements must be extensible. The architecture must be so modular, lego block like, that adding new functionality, storage or processing capacity is easily achievable with minimal (ideally zero) downtime.

As the problem morphs, the solutions must evolve to continue to remain relevant. The 'API-first' design approach enables solutions to evolve and new customisations to be built at speed. Designing a software element by first envisioning how it will interact with others ensures its usefulness to the co-creators in the network. The data model or schemas then follow from how the users use the data as opposed to what the system wants to collect. If the infrastructure elements are atomic and recombinable via clean interfaces/APIs, building new elements that plug into the system and interact with existing elements becomes easy.

The 'plug-in architecture' or the 'publish-subscribe' design patterns are some examples of extensible designs. In addition, building the platform using off-the-shelf commodity hardware also allows the platform to evolve as the underlying technologies evolve.

Another great example of extensible and atomic element is the Wordnet inside EkStep[2] infrastructure. It is atomic in that it serves a single purpose - allowing words across different languages to be retrieved along with their relationships, characteristics and sounds without worrying about the purpose of that retrieval. It was initially built out for antonyms, synonyms and relationships for words across English and Hindi, and later extended to 14 other Indian languages. Next dimension of extension added was hyponyms and hypernyms. Another possible future extension would be translations across these languages. The element is designed as a graph with clean interfaces that separate the logic around the graph and a set of APIs for external usage. The wordnet is also extensible in that it can be used across Societal Platforms as well - for example, any mission around education may use and contribute to the richness of a single wordnet. A separate one may be setup if the value amplification expected from cross-pollination of words is low. For instance, all healthcare related missions may have a separate wordnet and so on. It needs to been designed to scale with depth of use as well as adjacency/breadth of use.

[2]Societal Platform for Education (read more at https://ekstep.in/)

Refactorable

Enforcing a litmus test of 'how comfortable are you to let a new developer refactor your code?' will ensure due focus is given, early in the development cycle, to designing for extensive refactoring. Besides atomicity and clean interfacing, refactoring also requires appropriate levels of encapsulation, extensive support for automation, regression testing, devops etc., and a mindset of building for external as well as internal consumers.

Generalisable

The more an architect can abstract from context and mission specific use-cases and build as context-invariant elements, the more powerful the element becomes. Of course this would have to be balanced against the constraints of mission and its resources.

The possible generalisations may also emerge over time as more missions come forth with similar requirements. The design should be highly atomic, extensible and refactorable to support cycles of customisation and generalisation. This will yield powerful context-independent elements over time, that are then usable across contexts.

Thus the platform evolves as a wave with repeated iterations of customisations being distilled down into abstractions, new customisations rising from that and so on.

For example, the Registry Framework used in platforms such as EkStep and DIKSHA[3], evolved from being an aggregator for teacher data from different educational institutions to a general-purpose data-aggregation engine that can streamline access to different kinds of data and/or entities. It is a graph-based schema-less database that amplifies the usage of the data while giving the data owner control over it. The generalised design makes it usable for any data, aggregated from any source, for any purpose.

The tradeoff questions are, 'how much intelligence should we build into the system?' vs 'how much agency we want to leave with the participants and users on the network?' If an element is extremely abstract or generic, the most basic use-case will also need additional development effort and customization. However, if it's too complex with significant intelligence built in then it restricts the number of ways others can use or extend it.

The design should be highly atomic, extensible and refactorable to support cycles of customisation and generalisation.

The tradeoff questions are, 'how much intelligence should we build into the system?' vs 'how much agency we want to leave with the participants and users on the network?'

[3]DIKSHA (read more at https://diksha.gov.in)

Iterative

The technology elements that can be used across contexts and even platforms will not all evolve to perfection on day 1. It is important that they be designed to support iterative evolution. Along with atomicity and extensibility, they must also be upgradable and backward compatible. Extensive configurability is another way to provide external hooks to control system behavior and enable changes over time.

The architect may weigh these in the light of the realities of the ecosystem that will use this platform, from co-creators to end users. Questions like, what is the minimum viable first version? What parts can be iterated over? What is the technology know-how and expertise of those interacting with an element? Will their agency be restored or taken-away by additional complexity or configurability? Are the users capable of taking on the additional cognitive load that comes with added agency? Can we introduce this complexity in a phased manner? Are the devices capable of additional processing burdens? Is the infrastructure on the ground capable of supporting additional complexity? For example, an app based on native android that requires a user to initiate upgrade may miss out on critical security updates, thereby risking the individual and maybe even the network. Another example, if an app remotely manages storage between the end device and a cloud hosting, it can leave the user frustrated with a varying usage experience. It could also make matters worse by completely denying usage in a poor connectivity region. In both these cases, a tech-aware user may regard these as a loss of agency!

Additionally, how easy will the upgrade to additional features be at scale? Does upgrade require users to repeat already committed steps (relogin, reauthenticate, refurbish data etc.)? All these and more will determine how iterative an element is (or can be) and how the iterations play out with current set of users. Some of the supporting frameworks for iterative design, such as Versioning, Upgrade and Rollback, Testability etc., are elaborated below.

Versioning

Open source software with an increasingly complex set of use-cases and users requires versioning at a very granular level. Every element that is external facing (as APIs or as Datasets or as code) needs to be versioned separately and have clear compatibility matrices associated with it. Careful thought should be given to find a way for different elements to inter-operate across versions.

Upgrade-Rollback

A Societal Platform that touches millions or billions of end users, must have a well defined and easily executable upgrade and rollback path. The daily usage and multiple touch points that a solution built over this platform will have, forces careful thinking around how bug fixes should be released to the field, how security updates may be mandatorily pushed or how new features could be designed as opt-ins. These would classify the updates as mandatory or optional and be delivered via scripts or installers or humans.

Backward Compatibility

It would be natural to expect different versions of different elements interacting with each other across the entire usage spectrum. This implies that backward compatibility and appropriate deprecation cycles will need to be called out upfront, aggressively communicated to affected parties, appropriately handled in software along with building in the ability to run different versions of the software/APIs.

Configurability

Additional configurability makes a system highly reusable, but also adds layers of complexity and effort in making it usable! Additional configuration can provide agency, but can also take it away! After weighing these considerations the architect may determine that configurability increases iterative-ness but reduces speed of adoption or scalability. However, documenting this thought process will help the ecosystem of co-creators engage effectively. Once the set of configurations is determined, easy ways to enable mass-scale rollout may be envisioned for faster adoption. For example, predefined configuration templates that can be remotely executed at upgrade, or defaults for all non-mandatory settings etc. may be supported.

Automation

For all of the above characteristics to succeed at speed, special focus is needed on the ease of access, development, testability and deployment for all elements within the platform and also those built over it. Simplifying the upgrade process so that it can be performed without manual interventions, or automated handling of typical exceptions occurring in the system, or ensuring automated regression testing to ease refactoring, all become significant design goals for a system at scale.

Testability, via automated mechanisms, is also key for the Societal Platform ecosystem to thrive. To illustrate, if a developer cannot easily develop, drop and test a new plug-in, the network of participants withdraws and the pace of context intensive customisation drops. Enabling individual testers thus becomes significant for evolution at speed. Or, for example, if regression testing is not automated, developers become reluctant to experiment and refactor and participation of builders and extenders drops.

Designing for Scalability

Responsive to Scale

Designing for scale from day 1 requires architecting the underlying elements such that performance and capacity are modularized and capable of responding to increasing demands. This does not imply building for the full-scale from day 1, but to design for the right scale and elasticity such that the code can be optimized for a single server and also effectively scaled for distributed set of servers.

Specifically, the context-invariant elements must have the ability to:

1. Handle increasing number of user interactions / transactions over time
2. Handle sudden unexpected load surges and corner cases in runtime
3. Increase storage capacity and processing capacity over time

Building the platform as a set of microservices delivered over a multi-tenancy architecture would be one of the ways to enable these abilities.

The processes and frameworks that are heavy in user interactions or dependent on user actions will also need to plan for large-scale errors as the number of humans interacting with the system increases. For example, a content framework that relies on humans/users to tag content but does not pre-classify or provide a set of tags may fail at scale as the diversity of tags increases with the diversity of humans doing the tagging. Over time this will adversely affect the discoverability of the content itself and result in a poorly scaled content framework.

Reliable and Resilient

Reliability and resilience are integral to any large scale digital infrastructure design. However, in the case of Societal Platforms, they take on added significance from the start. Given that human touch points will scale rapidly and suddenly, consistency of system

behavior at one and one million transactions becomes imperative. Thus reliability or dependability of the services has to be baked into the platform infrastructure. The system has to self-heal as far as possible, and while this may not happen on day 1, the path to it must be considered.

This requires careful consideration of failure paths and a design that minimises the number of experts needed to fix a failure. Exception handling should be baked in, at first using right events/data points to capture a failure, thereafter by providing tools to analyse the events and raise effective alarms, and then algorithms to understand and respond to a failure. At scale, even the edge error scenarios can bloat and drown the platform. Phasing out the handling of different exceptions (some only monitored, some analysed/reported, some self-healed from day 1, and so on) enables the architects to respond appropriately to a scaling system. For example, in a financial transaction system such as micro-lending or DBT, a failed digital transaction that cannot be identified, isolated and rectified promptly has severe implications to the consumers. Assuming a failure rate of 0.01%, at 1 million transactions a day that's 10,000 people and their monies! A manual intervention to problem solve is not sustainable or scalable, and hence the need for systems to self-heal.

Given that human touch points will scale rapidly and suddenly, consistency of system behavior at one and one million transactions becomes imperative.

Usable by All

Many software and hardware design decisions can easily include or exclude whole sets of audiences based on how they access and understand technology. Choices with respect to access and availability should enable a mission objectives as well as future growth.

Levels of tech expertise of audience, access to internet connectivity, languages spoken, disabilities if any etc., are some of the details that need to be considered upfront. Availability across a variety of devices (laptop, mobile, POS[4] devices etc.) and support across softwares (OS, browsers etc.) also become important factors in making the platform available to all. For example, choices such as limiting the delivery to be mobile only, or online only, or specific OS or web-app framework all exclude swaths of populations and collaborators. Such choices, early in the life of the platform, can severely restrict growth later. For example, the EkStep platform consciously chose to use HTML-Javascript for its applications over building a Raspberry Pi

[4]Point of Sale

only solution. This enabled a wider ecosystem of collaborators and users, and removed the burden of ensuring device penetration on the core elements.

In addition, the architect may also consider the realm of future devices and man-device interactions. Can the design lend itself to taking advantage of those advances? For example, a learning science tool should be independent of how the content it provides is accessed; its behavior is independent of its invocations, be it via audio, video or text.

Designing for Trust

Open

Open Societal Development is essential to designing for trust (and thereafter, scale) on a Societal Platform. It enables transparency (w.r.t. governance, but also w.r.t. 'being what it claims to be'); as well as collaboration. It comprises of two equally important aspects:

1. Reusing existing open source software or elements from other Societal Platforms
2. Enabling others to reuse, extend, and create solutions over what you build

Openness is possible along various dimensions and all are relevant for Societal Platforms:

1. **Open Source:** Making the source code available to public via repositories like GitHub establishes trust in the builder ecosystem and enriches the design itself.
2. **Open APIs:** Open APIs enable ecosystem players to build additional services and extensions faster, leading to faster contextualisation and solution delivery.
3. **Open Architecture:** Using open standards and frameworks ensures the design is vendor neutral. This allows the software and hardware of the platform to evolve as the standards themselves evolve.
4. **Open Data:** Opening up the data on the platform basis constraints of ownership, consent and privacy, allows faster decision making and necessary corrective actions.
5. **Open Governance:** Discussed under Transparent. (Refer to page no. 14)

The openness to use, extend, adapt and contribute back to the process of open societal development is key to enabling scale and speed. As is the ability to let go of one's design for others to use, extend, adapt and customize as they see fit. Another important aspect of Open Development is the foresight to enable contributors from the tech community via active engagement of the ecosystem (discussed under Consumable). (Refer to page no. 15)

The openness to use, extend, adapt and contribute back to the process of open societal development is key to enabling scale and speed.

Consistent

Consistency of system behavior via well-defined handlers and paths, as well as consistency of communication with the ecosystem of builders, extenders and participants, will build trust on the platform and for the platform. While some of this will get implemented as user policies, the technology elements and the engagement ecosystem must also reflect these in action. For the architect, the focus primarily should be to ensure transparency in logic and outcomes and consistent system behavior at all times.

Democratic (Governance on the platforms)

The existing ecosystem that is attempting to solve a societal challenge and the individuals affected by the problem are the key stakeholders of a Societal Platform. The platform must be designed to provide each such participant an equal voice and the ability to come together as a community. This may be achieved by actively preventing single points of control within the user-interaction flows, and preventing power structures or control points from forming by virtue of most common user interaction paths. The need for moderators, elected by the community may also be supported by the platform.

The goal should be to trigger highly federated control within the community and the platform.

The goal should be to trigger highly federated control within the community and the platform. Some ways to achieve this are maker-checker frameworks to ensure multiple parties sign-off on a transaction, reputation engines that allow users to vote up/down, authenticating features that prevent anonymity, flagging frameworks as a system to raise alerts faster etc. Blockchain is also regarded by some as the future of federated control in collaborative networks.

Transparent (Governance of the platforms)

In order to truly realise the potential of open societal development and co-create the platform itself, the choices with respecting to hosting, documentation, licensing policies, and issue resolution must also be well thought out and transparent. The architect may consider tracking the issues and priorities openly on a public forum.

Co-creative
(Co-creation on the platforms)

The platform must actively encourage co-creation through collaboration and communication, learning and sharing. Thoughtful design and technology elements can bring forth this outcome. Some examples of this are elements that:

1. Promote easy onboarding:
    a. Keep the entry barriers really low to support diverse co-creator needs - some simply consume/learn, some review, some add/extend etc.
    b. Support different formats of upload, diff formats of viewing/downloading, support different browsers, devices and so on.

2. Promote learning and sharing of knowledge
    a. Blogs, powerful content search
    b. Gamified indicators such as leaderboards, badges, unlocking levels etc.
    c. Showcase innovations and positive/powerful contributions

3. Promote communication
    a. Community Setups/management
    b. Message boards/forums
    c. Recommendation engines
    d. Notifications/mailers etc.

Consumable
(Co-creation of the platforms)

In order to truly realise the potential of open societal development and co-create the platform itself, the design elements and the design process must be easily consumable by the tech ecosystem. This may require extensive documentation (in code and otherwise), effective handling of error paths and error codes, as well as active community engagement, mobilisation and support.

The three platform actors will have different needs and require different kinds of engagement and support. While these do not constitute a technology decision, they are critical to a Societal Platform's speed, scale and sustainability aspirations. The architect will thus need to envision the modalities for these engagements and support and create that DNA at buildout.

Instrumentation and Insights

To be transparent and iterative in evolution, the platform design must embed extensive analytics and also analysis. If all stakeholders are able to view these analytics transparently, it should lead to better insights on shortcomings, wiser enhancement choices, better metering and throttling of usage (if required) and so on. A strong analytics engine and consistent set of metrics tied to purposes will over time reveal insights that can feed back into the platform as features or policies.

Event Framework

The architect should lay ground rules on the events to be generated and captured by all elements and extensions on a platform. A generic event framework can enable this and ensure standardised telemetry collection across all elements. While we want the events generated to be pervasive to provide maximum transparency into the platform, it must be balanced with invasiveness of those events and their impact to performance. While every event or line of code has a processing overhead, goal should be to minimise this or manage it effectively on a live system.

Digital Trust

Digitizing “trustworthiness” of an individual or organisation or transaction or asset, (in other words, digitizing the “establishment of trust”) and making it available to the actors in a Societal Platform increases the diversity and inclusion possible on such a platform. Physical trust established by active human engagement, or via trust profiles established through paper trails, proofs and assertions is expensive and needs to be repeated at each new transaction point. Digital trust, on the other hand, may be established through identity (for individuals and organisations) and immutability (for transactions or assets) and can be machine verifiable. Digital trust established through such ‘trails’ can never be repudiated or modified, are in the possession of the data subject to whom they belong, and enable actors to actively engage over it. Such trails also allow the network to actively dis-engage with and reject malicious actors on the network.

Digital trust must span the actors and the transactions or assets they create, but should also extend to the interfaces provided by the platform itself. How does one element on the platform trust the input it receives from another element via the open interfaces?

Designing for Data Empowerment

Any Societal Platform, by its very nature of being societal, digital and large scale, will handle large amounts of data. Ensuring this data is effectively used to create value for the society and to restore society's ability to solve its issues, in specific contexts as well as at systemic levels, is fundamental to the Societal Platform way. While the philosophy/framework for this is described elsewhere, its translation to technology and design is briefly described here.

Data on a Societal Platform is an engine for social change, not a tool for accruing value to the platform. It must thus be managed and secured with a lens on ownership and visibility. This implies defining ownership and visibility of every data field and managing it via transparent rules and policies.

Data on a Societal Platform is an engine for social change, not a tool for accruing value to the platform.

The technology focuses on implementing data management and data security. While policy dictates the clauses under which data may be managed and secured, data privacy policies are subject to local laws. Hence the architect may also need familiarity on those context specific requirements.

Some questions to consider around data management and policy are:

1. **Data ownership:** Who owns what parts of the data on the platform? Do they have rights to withdraw that data at anytime? How do they exit the platform at will, taking their data with them? Do they wield any power on the platform by virtue of owning this data (Data Currency or Data Empowerment)?
2. **Data curation, consumption and purging:** Who adds data to the platform? Who consumes it? Who removes it and when? How do they do these activities? What policies regulates them?
3. **Data privacy and security:** What is the data privacy policy and how does that define the data security measures to be implemented via technology? How is consent established and propagated in the platform?
4. **Individual privacy:** How do you mask private info and shield individual privacy? How do you ensure secure access and guard against (+recover from) breaches, both at individual and systemic levels?

Some questions to consider for the data generated on the platform are:

1. If there are multiple sources of data, does the platform synthesise across them?
2. Who certifies which data is accurate? Who draws insights from it?
3. Can conflicting data reside on same platform? Does the platform need to highlight it? Can it *(technology debate)*? Should it *(ideology debate)*?
4. How can the data generated on the platform be more public/ open?

The treatment of data on a Societal Platform will set it apart from a for-profit platform and these design considerations are merely indicative of the direction an architect should take.

Summary

With the core values of Societal Platforms in mind, the builder or extender seeks to unlock scarce resources, create an environment conducive to co-creation, and support and enable solutions for diverse communities. As they embark on this journey, the listing of design characteristics serve as checkpoints to stop and reflect on.

It is by no means exhaustive or final. New learnings are expected, as is the inability to realise them in one go. The architect may make compromises or takes decisions that are non-ideal due to various constraints. The key will be to identify this technical debt as a burden on the platform and to establish processes to remove them over time.

Be part of the Societal Platform Collaborative by sharing your journey - your learnings and experiences with us. Write to us at info@societalplatform.org.

SOCIETAL
PLATFORM

www.societalplatform.org
info@societalplatform.org