HOT OSM Task Manager 3 Tech Challenge

Application: Nilenso Software LLP

Introduction

Throughout this application, we hope to address all questions and concerns outlined in the Task Manager 3 (TM3) Tech Challenge (TM3TC). Our team is familiar with HOT, has participated in several mapping events organized by the local OSM community, and is particularly well-equipped to handle both the technical and non-technical challenges of the TM3TC. Our company also has a very specific vision of progressively delivering more of our work as open source, preferably around open data, with each project we take on. When we evaluate, sell, and staff projects at nilenso it is rare to come across one so well-aligned as this. We are very excited to apply to the challenge!

We have arranged our application in a slightly different order than the format of the TM3TC document or the project Terms of Reference (ToR). Because it affects absolutely every aspect of the project, collaboration and communication is mentioned multiple times in both of these documents. We feel this is a key issue, and we have started our application with this topic and how we plan to address it.

After collaboration, we briefly describe our company's skillsets and then dive into our experience in, and approach to, project management, design, and development. We intend to staff people from each of these three roles on this project.

Finally, we will directly address how we will achieve the goals outlined in the TM3TC and present our company's qualifications for taking on such a project.

Collaboration

Nilenso is a boutique software consultancy based primarily in Bangalore (India). In our 3-year history, we have built large machine learning, streaming data, experimentation, and mapping/transportation systems. We continue to run the largest of these systems in prouction for our clients today.

No two customers are ever alike and we do not have a strict approach to collaboration with our clients' teams. Our work has moved us around the country and around the globe, operating in and across many different timezones, with many different project management styles, in a variety of toolchains[1]. Our default delivery style is based on the classic Extreme Programming methodologies with 1-week iterations, daily standups, TDD, pair programming, and heavy communication. However, our PM and communication has spanned heavier up-front

design, kanban pull queues, or (in rare cases) no process at all. Some clients prefer we speak to them as much as physicaly possible… some prefer to leave us with high-level technical requirements and check in every couple weeks.

What matters most in our collaboration with customers is not the particular strategy used, project management style, or tools chosen. The essential qualities of a successful project are open lines of communication and planning/projection which employs strict measurement and reduces guesswork. The ToR Methodology bullet "A." diretly addresses this first point by identifying the varied channels which stakeholders will be employing. To take this a step further, one of our first objectives would be to document all stakeholders (or as many as possible) to ensure the graph of communication was not only open along the channels themselves, but also transparent from the outside, as a whole.

Planning and projection demands a heartbeat of some sort, and even when using a kanban pull queue to prioritize work, some period (often one week) must be used to measure progress and project into the future with the same metric. Progress on individual features/stories/cards would be updated in real time (daily) so stakeholders can watch work in-flight. The one-week period would be a summarization window for work accomplished, since daily progress is usually too narrow a window to make meaningful predictions.

See: TM3TC "Scope of Work" 1. Collaboration

[1] We are intimately familiar with most of the project management tools out there: Pivotal Tracker, Mingle, JIRA, Trello, paper-stickies-on-a-wall, and of course, Github.

The other side of the reporting and planning coin is, of course, the approval process for new work. The TM3TC is somewhat ambiguous as to whether anyone from the HOT community will take on the responsibility of owning final approval and prioritization or if this will fall to the development team. This is one area where we do strongly feel an authority is required, even if this person (or people) act as a facilitator rather than a gatekeeper. A group approval and prioritization process is possible if it is executed in real time, and we would strongly recommend one such process which we have used to great success on many projects: a project *Inception*.

Inception

An Inception is a high-intensity kickoff to a project which captures a shared understanding of the high level goals, their priorities, their budgets, and their limitations. The remainder and majority of the Inception then attempts to capture features (or "user stories", preferably, as these frame each feature in terms of how it benefits the end user), estimate them if necessary, and prioritize them. Whether stories are individually estimated or not, they are then packed into a few sample weeks of the project to rough out the expected pace of development.

With this initial projection, 4 months worth of TM3 stories can be laid out in priority order and we will know roughly how much the team feels they can accomplish in that period.

Stories can be reprioritized as the project progresses and the initial projection will be updated every week with the team's actual rate of progress. The project's scope is adjusted every week accordingly.

While not every project we execute begins with an Inception, there should be some form of project kickoff which ensures stakeholders are in agreement about what they hope to achieve within their budget and timeframe and what the development team anticipates they can deliver. Inception or otherwise, this initial project kickoff should leave the development team confident they clearly understand the deliverables and dates – and leave the stakeholders clear and unambiguous about high-level goals and what they are likely to see at the end of 4 months. If either side is confused or unhappy at this very early stage, we would go back and clarify or re-evaluate the project. We do not start projects without the confidence we can deliver what is being asked of us.

During the project, new feature requests can come from the stakeholders through the various channels into a structured approval and prioritization process. If all cards/stories are ordered by priority from the beginning, it will be easy for the team to collectively prioritize (or for an authority to individually prioritize) individual stories against that total ordering.

See: TM3TC "Scope of Work" 2. Proposal / Approval of feature implementation

Another important aspect of collaboration will be that of nilenso with any other contributors from within the community. As TM3TC encourages individual community contributors to apply, we similarly want to encourage a mixed-mode award of the contract. We would be more than happy to work with a contributor(s) who has more experience with HOT or OSM in general, either as an active community member or active contributor to HOT tools.

While we are applying for the entire Scope of Work (SoW), we happily acknowledge that this may not be possible or optimal.

See: TM3TC "Scope of Work" See: TM3TC "How to Participate" 1. Identification, 3. SoW Fit

Our collaboration strategy involves user testing and user feedback of new features, solidifcation of existing features, User Experience, and design in general. While known stakeholders can be actively communicated with, one of the most important goals of TM3 is the onboarding of new Mappers. The design for a new Mapper begins outside the scope of the project itself: How did they hear about OSM? How did they learn about HOT? What do they hope to achieve? How easy/difficult is it for them and how long does it take? How can we make that entire transition smoother from the very beginning?

These questions are best addressed by our Design team and on a continuing

basis. Passive feedback may involve providing an easy feedback mechanism through HOT and OSM channels, within the website or within the app. Active feedback could involve cafe testing, explicit new user solicitation exercises, and surveys.

User Experience feedback from experienced Mappers, Validators, and Project Managers will be easier, as we can reach out to them directly and afford them multiple channels to reach the TM3 team. The team can then centralize and collate this feedback and look to formalize changes (in the form of stories) based on problems identified or positive UI/UX/documentation patterns users would like to see repeated elsewhere.

One consideration for this latter case is how such an open feedback system will work once the 4 months of the TM3 project have ended. If the feedback mechanisms are too open or too ad-hoc for the community to manage, we may need to consider a more structured approach, such as a single-channel "report an issue" feature within the app itself or Github issues (though these can be offputting for less technical users).

See: TM3TC "How to Participate" 7. UI Testing and UI Feedback [Note: "UI Testing" and "QA Testing" are separate, and largely unrelated, topics and we address the latter below, under "Development"]

The last topic to address within the scope of Collaboration is scalability, sustainability, and community engagement. Scalability from an architecture perspective we will leave until the "Development" section, below. However, scalability and sustainability are also questions of collaboration with respect to language, environment, and ecosystem choices.

It is quite clear that a substantial portion of the OSM and HOT software ecosystem runs well on JavaScript and Python. TM2 is no exception, and although under "Development" we will propose two alternate approaches to the tech stack (continuing with Python or replacing the Python backend with one written in Clojure), as far as collaboration is concerned we have a few issues to address:

First is the community's appetite for alternate technologies. If the HOT/OSM community is, generally speaking, more comfortable with widely-adopted technologies, we would hate to jeopardize our application by insisting on a language with a narrower user base (even if the JVM framework itself has a much larger following).

Second is the burden we might create by adding a new language and framework to the mix. Again, we wouldn't want to excessively slow down future progress if every new contributor to TM3 had to learn Clojure from scratch.

Last is the more positive spin on the possibility of writing TM3 in a LISP on the JVM. A LISP affords a much more disciplined approach to functional programming than Python. Though the value of statelessness, immutability, concretized notions of time, and an STM can be partly captured even in object-oriented and dynamic languages like Python, Clojure provides very clear delineations

between pure functions and those which produce side effects. The JVM provides a plethora of advantages, from libraries (though the Python is generally healthy in this regard) to tunable/swappable garbage collectors to plug-and-play analytics and production monitoring to pluggable profiling tools. Clojure's ecosystem has also quickly grown to be one of the most mature with respect to testing, documentation, and low-dependency modularity. All of these features make Clojure a surprisingly accessible language, even for programmers new to the environment, and an incredibly *productive* language for those who are even somewhat familiar with it.

Skillsets

Open source has become the default mode of operation for much of the software industry, and many of us have begun to take "openness", in the form of source code at least, for granted.

Nilenso is in a unique position in that we are an employee-owned co-operative. While we are still a profit-making corporation, as a knowledge worker co-op we have a vision and direction which is collectively and collaboratively defined by our employees (members). Collaboration and cooperative decision-making were the very genesis of our organization. Thus, when nilenso states that we want to spend all our time building open source software in a collaborative environment, an observer can be certain that participation in Open Culture is not just a means to an end, but the organization's modus operandi.

Nilenso offers a collection of skills which will all dovetail nicely on this project. Though we are primarly a software development company, and the majority of our members are software developers, we also have a decade of project management experience and a skilled design team. All aspects of our team's skillset influence, and are influenced by, open systems: Our project management strategy is entirely about transparency, open communication, and reducing risk by failing fast. Our design team is not only familiar with the OSM ecosystem, participants of HOT, and dedicated tree mappers of Bangalore – but they understand and have experience working with open data systems. Openness and progress go hand-in-hand, and designing instruction-free, intuitive systems are their passion.

Our software developers are the reason for the majority of our acclaim, however. Though nilenso has only existed for 3 years, we have decades of experience delivering modular and abstract code which has evolved with the industry over the past two decades. "Reuse" for us is much more about clean design – well-defined service boundaries, appropriate library usage, and logical abstration – than it is an goal unto itself. Software and team performance both are consequences of a clean design, not the other way around. Our recent technical achievements can be read about here: http://nilenso.com/recent-tech.html

We'll next take a look at these three disciplines in order of increasing specificity: Project Management, Design, and Development.

See: "Evaluation of Proposals" 2. Reuse of Code

Project Management

In order to address the TM3TC request for "A timeline overview of the expected major activities and milestones over the 4 month period" ("How to Participate" 2.), we provide a sample set of stories and a general outline of major miletstones after that. from the requirements we've gathered from the TM3TC and the ToR.

We have intentionally *excluded* most of the nearly 300 Github issues for this exercise. Though the Github issues list provides some fantastic examples of work the project will want to prioritize, particularly given those very incisive and clearly defined issues (ex. "Prevent editing until 'instructions' page is read": https://github.com/hotosm/osm-tasking-manager2/issues/890), the list is not consistent in its entirety. The Github issues contain many duplicates, wide-ranging size and difficulty, submissions from users and administrators of all experience levels, and it is difficult to know which issues are fresh/stale/old-but-very-important. Scrubbing all 300 issues in an attempt to come up with an exhaustive list of stories without any stakeholder participation will quickly provide diminishing returns.

Instead, the high-level requirements from the TM3TC and the ToR (as well as conversations from #task_manager_3 in Slack) provide a handy sample-sized story list which is still representative of the project goals on the whole. A few Github issues have been included in story format. It should be noted that this list is in no way exhaustive and does not assume build-out of TM2 or a from-scratch server-side rewrite. Therefore, a reader may notice some stories which describe functionality TM2 already provides.

Stories

The sample story list is as follows. Non-functional requirements are prefixed with "[NFR]".

> *NB: Since one of the Github issues listed is "Consistency in terminology: tasks, tiles, and squares" (https://github.com/hotosm/osm-tasking-manager2/issues/912), we will use "tiles" below to mean all three.*

- Validator can query list of hints in TM3 of tiles where OSMA anticipates an error/mistake
- Mapper will be automatically notified when OSMA anticipates a common mistake on the tile the Mapper is currently editing

- PM/Validator can see # of changesets for a Mapper
- PM/Validator can see # of complete tiles for a Mapper
- PM/Validator can see days since account activation for a Mapper
- PM/Validator can see # of days of active mapping for a Mapper
- PM can view an aerial map of historicaly, current, and new project space by geography
- PM can view a map displaying active and historical projects
- PM can clone a project to create a new project with defaults
- PM/Validator can observe changes made by a Mapper in real time to provide early feedback
- Mapper can self-assign a tile to herself
- PM can assign a Mapper to a tile
- PM can mark a Mapper with a "Validator" role for a project
- PM can create a project group
- PM can add/remove projects to/from a project group
- PM can see current and historical project groups
- PM can mark a project group completed
- PM can mark a project completed
- PM can see nearby/intersecting projects on the project creation screen
- PM can see nearby/intersecting projects on the project summary screen
- Mapper can provide in-app feedback about aspects of mapping they find confusing
- Mapper can make a generic request for help to a Validator
- Validator is notified when a Mapper is identified by OSMA as needing help
- Validator can see a prioritized list of tiles requring validation
- Validator sees likely error-prone tiles highlighted by OSMA in the prioritized list of tiles requiring validation
- Validator can send Mapper quick feedback from a predefined set of common issues
- Validating a tile acknowledges the Validator for that task
- Validator can mark a tile validated in JOSM (implies corresponding API)
- Mapper can optionally add an email id to her profile
- Validator feedback is sent to Mapper's email, if available
- [EPIC] TM3 should provide an API for custom analytics queries (See ToR note "Laura O'Grady 12:38 PM Nov 24")
- PM can set expiration period on a tile; tile auto-expires at the end of the period
- PM can assign a difficulty to a tile
- PM will see auto suggestions from OSMA when assigning difficulty to a tile
- Any User can see aggregate analytics on tiles for all of OSM
- Any User can see aggregate analytics on tiles for a project group
- Any User can see aggregate analytics on tiles for a project
- Any User can see aggregate analytics on tiles for a user
- PM can see a list of Mappers for a project

- TM3 should support 100,000 simultaneous users on XYZ hardware (this card may be defined in terms of N users per M hardware nodes, assuming performance tests are published displaying linear horizontal scalability and the failure conditions for horizontal scaling)
- [NFR] Any API user can read a live data stream of activity from TM3
- PM can view a summary of projects & statuses by geographical area
- PM can adjust tile size to accommodate varying complexity/difficulty
- [NFR] Finish pending TM3 API endpoints for existing features (See TM3TC "Goals" 5.)
- Validator can ask a question of a Mapper for tile she is validating
- Mapper can view a TODO/checklist for her current tile
- Mapper can see a prioritized list of task squares available to work on
- [NFR] TM3 must support i18n
- [NFR] Any user can submit a translation or update to a translation for review
- [NFR] Mutating API calls must be authenticated
- Mapper/Validator can request access to a private project
- Mapper can mark a tile as "too cloudy to complete"
- Mapper can mark a tile as "imagery insufficient to complete"
- Mapper must review & certify TODO/checklist complete to mark tile as "done"
- Mapper can stop working on a tile to unlock it without marking it "done"
- Any User can subscribe to notifications for all comments made on a tile
- Mapper receives a "Welcome to HOT mapping" email upon first login
- Mapper receives a buffer around tile area when beginning work on a new tile

Those unfamiliar with User Stories and their place in XP or lean methodologies should be aware that a story is not a requirement or a feature. The best description of the one-liners above is that they are each a "promise of a conversation". Stakeholders and developers will both understand what each one means at a high level but delivery will require that the development team engages directly with the stakeholder(s) with the most expertise on any particular topic to ensure they are building the software everyone wants. This is then validated and refined through a user-owned QA testing process in the HOT/OSM testing environment before features are released into production.

Timeline

Prioritizing stories and NFRs (collectively, "cards") will be a collaborative effort with key stakeholders. If high-level feature groups interleave priorities, "milestones" may simply be markers showing overall progress. Only if priorities and groups of features line up will thematic milestones emerge; we would recommend against forcing the concept of milestones into the delivery schedule as they only provide observers with emotional satisfaction but do nothing to structure a

project more effectively. Prefer continuous delivery.

The cards in the above list are not of a consistent size. "TM3 should provide an API for custom analytics queries", for example, is marked "[EPIC]" because it is substantially larger than all the other cards in the list. Before delivering such a card, it would need to be broken down into relatively similar-sized cards (to each other and to all other cards in the list). If the team finds it difficult to apportion cards in roughly similar sizes, cards may be given relative estimates, in abstract points: 1, 2, and 3. "1" describes a nearly-trivial task. "2" is the preferred difficulty for most cards. "3" represents work which cannot be subdivided into cards of size 1 or 2.

For this exercise, let's assume we were able to apportion cards of roughly equivalent size for all 58 cards above. The aforementioned API epic, "Validator can mark a tile validated in JOSM", "TM3 should support 100,000 simultaneous users", and "[NFR] TM3 must support i18n" are all obviously larger cards. Let's assume these 4 cards are broken down into 16 smaller cards we consider roughly equivalent to the rest of the list. If we provide the prioritized list to the software team, they can then plug cards into the first few weeks of development to estimate their week-on-week accomplishments. Say the first week fits 5 cards, the second week 7, and the third week 6. The sizing of effort in this exercise is fine-grained enough to be possible for a human being to estimate but coarse-grained enough to be meaningful on the 4-month project timeline. Our 70 cards will take 12 weeks to complete, leaving 4 weeks buffer for scope creep, underestimation, manual QA, and inefficiencies in the overall feedback process.

Obviously the real Inception exercise will be much more involved and collaborative. We hope this provides some insight as to how we would go about it, however.

Regarding "How to Participate" 6. ("At least one specific feature addition you propose…that would have the largest impact…"), given that it is our aim to break features down into actionable tasks as early as possible in the process, it becomes difficult to identify one particular feature-set which has the "largest impact." As previously mentioned, it's possible key stakeholders would prioritize individual cards in such a way which interleaves overarching feature-sets.

With that said, it appears there are a few areas which would be quite impactful:

1. Exposing all TM3 behaviour by auth-based APIs, first and foremost, will allow all participants to to manage projects & groups and execute tile-level work from outside of TM3 and within their preferred OSM editing environment. Though elided from the cards above, new authenticated mutative APIs will require significant work from the development team in tools like JOSM to ensure a full spike through the system, from the user's tool, through TM3's APIs, into project data. The best user interface feels like none at all.

2. User- and Project-level analytics/reporting for Project Managers and Val-

idators will help greatly in informing their workflow and making management of projects more efficient both by improving meaningful (informed) communication with Mappers and helping to smooth the Validation process. The Github issues list contains much low-hanging fruit in this regard and the community's pain points should be our starting position for quick development.

3. The most direct and TM3-specific feature-set will be interfaces for project/group creation and administration. Though much of the Mapper- and Validator-facing functionality will hopefully move to tools on the edge of HOT/OSM architecture via TM3 APIs, projects and groups of projects are concepts which are rooted in the Task Manager and the unified UI is likely to continue as a feature of the web app. While the primary goal of the user experience for Mappers will be ease and elimination of friction, the goal for Administrators and Project Managers will be power and flexibility. Administrators are not new to OSM and do not require "conversion" to the process; the PM interface of TM3 may forsake time-consuming investment in slick UIs and an easy user experience for richer, heavily-documented features.

See: "How to Participate" 2. A timeline overview See: "How to Participate" 6. Specific feature addition(s) proposed See: "Evaluation of Proposals" 5. Clear examples supporting HOT projects

Discovery

Beyond an Inception (or other form of project kickoff), the discovery process will be daily and continuous. We could run weekly Iteration Planning Meetings with stakeholders or speak to them day-to-day, depending on what works best for various community members' availability. Because of nilenso's offshore and distributed history, we are very familiar with asynchronous communication channels, discussing issues and ideas over a multitude of channels. Even on commercial software projects in a full-time, proprietary environment it is highly unlikely that any two individuals will communicate in the same way. Though an increasing number of our customers prefer Slack (or IRC, or another equivalent) to email, many forums and mailing lists remain much more active than their semi-asynchronous counterparts. We've found for discussing specifics of a particular implementation, it's often best to negotiate a date and time for a video chat with shared screens. Our team works a long and staggered workday starting from 8:00am IST with the last folks usually closing the office around 8:00pm. Everyone makes room in their schedule for calls from home in the early morning or late evening to accomodate the "worst of all timezones" (at 11.5 or 12.5 hours difference) for collaboration with India: California. All other timezones are slightly easier to find compatible call times on one end of the day or the other.

What will potentially make the discovery process difficult is not the communication channels themselves but identifying authority figures in the community, facilitating conversation across more than two timezones, and quick turnaround with volunteer community members where availability won't be consistent. We look forward to charting and documenting the graph of stakeholders and community members as early as possible to identify challenges, dependencies, and blockers. For example, if a medium-priority card is best discussed with an expert volunteer who is only available sporadically (say, twice a month), we may increase the priority of the card to engage with them earlier in the process. Alternatively, such conversations could be managed up-front, eschewing some of the real-time preferences baked into lean and agile principles in favour of resolving blockers in the dependency graph.

The nilenso office is around the corner from the Mapbox Bangalore office. We would plan to lean on our relationship with (and proxmity to) the Mapbox team to resolve issues quickly. Value would always be dictated by the community, collectively, and an authoritative voice may need to come from the community at a later date (or even the next day) but we feel the Mapbox team's realtime expertise would be invaluable.

See: "How to Participate" 4. Working with community members / Feedback

Design

The nilenso Design team will participate in the development of TM3 with the aim of answering all 4 major user-facing goals laid out in the TM3TC, but Goal 3 is of utmost importance: "a significantly improved UI/UX" means progressively working toward the core of the Task Manager by first eliminating any pain experienced by users. Once the periphery of pain, discomfort, and confusion are eliminated we can then find the real heart of TM3 and its ecosystem and bring this to the surface.

What does this mean? We all remember our first HOT mapping party. Though OSM is relatively easy to get moving with for a technical user, it was never the case that a first-time HOT Mapper would sit down with her laptop, download all the required software, create the required account(s), and immediately begin working on a tile. Every time a new Mapper hits a snag, feels confused, experiences a bug, or has to ask the resident mapping expert for help, the process has failed her. Because these difficulties are repeated and well known, they will be easy to codify as a natural part of the user experience. This is the low-hanging fruit.

Next, we can move on to the core of what HOT and volunteer OSM contributions represent: relentless improvement. The quality of this improvement spans two spectrums. The first is, of course, the data in OSM itself. Humanitarian aide workers, governments, and citizens on the ground are better off for every single accurate node entered into the OSM database. But the spectrum relevant to

TM3 is that of *individual* improvement. Once a new Mapper is up and running, how can we make her a master of the craft as quickly as possible? Designing for a multitude of technical proficencies, time committments, and learning styles means designing an *educational* system for these individuals. This is no trival task, and our Design team relishes the opportunity to solve problems in the education space using data, analytics, simpler UIs, and intelligent feedback to users.

Perhaps somewhat unintuitively, the most substantial major shift in computer usage of recent decades to influence such an environment is mobile. As designers have had a chance to work with limited hardware and limited input, these constraints have taught us, as an industry, to build better user experiences which offload as much of the "thinking" as possible to the system itself. This may be a lofty goal for HOT infrastructure and OSM in general, but it's a challenge we would love to take on.

Our designers will be key in ensuring the development team is building what users need and what the community wants. They will participate in discussions about improving experiences, like those mentioned above for Mappers, for all HOT roles. The proposal and approval process will involve designers early on, from Inception to the fleshing out of each user story. During the Inception, designers will facilitate discussions around a cohesive, overarching design. When defining each user story, designers will describe the specifics of all physical aspects of the software: appearance, interaction, and time.

Designers will own the conducting of user experience testing, whether that happens in the HOT testing environment online or in offline testing such as cafe tests for new users who are completely unfamiliar with HOT, OSM, and their respective software ecosystems. Discoveries will be documented alongside the Architecture Decision Records (described below, under "Development") and maintained as part of version control history.

See: "Scope of Work" 2. Proposal / Approval See: "Scope of Work" 6. Usability testing See: "Goals" 3. A significantly improved UI/UX

Development

Documentation, Automation, Testing, Deployment

We will describe our development team and our successful history of project delivery in more detail at the bottom of this document, under "Who/What is nilenso?" However, to address the development-specific concerns from the TM3TC:

All of our software is thoroughly documented. We not only create user documentation (which is then validated by users as each individual new feature is continuously delivered into production) and developer documentation in

the form of in-repository documents, in-line comments (where the domain, abstractions, or algorithms may be confusing), and thorough test suites. We also strive for self-documenting code, verbose and meaningful commit messages on atomic (transactional) commits, short-lived branches, a master branch which is always green and deployable, and a continuous integration (CI) system which not only alerts developers to broken builds but provides proactive insight into deployment and environmental failures. Major architectural decisions will be captured within the version control system as Architectural Decision Records (ADRs: http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions) in an immutable, linear fashion. Every software project should contain a README which serves as the *only* point of entry required for a user, sysadmin, or new developer. Where documentation is not possible (for passwords, keys, and other secrets) the team members responsible are called out explicitly in either documentation – preferably within the README itself – or through automation tools at the point where interjection is necessary. A new user (playing a "sysadmin" role) should be able to install any piece of software with a single package or script for installation. With respect to operations in canonical environments (say, http://hotosm.org) this automation should include provisioning and orchestration.

Comprehensive load, integration, acceptance, and unit tests are a good indication of a healthy software project. In recent years, we've started to grow away from these practices to some degree. Load testing can often be completely automated and with small, simple, custom tools load testing can become a part of CI or even Contiuous Deployment (CD). Integration tests should not be a one-way street and within an integrated service architecture, client-driven service contracts provide integration tests which not only ensure narrowly-defined API correctness but facilitate communication and provide early alerts to both parties when API contracts are accidentally violated. Complex systems, such as the Data Science Experimentation Platform (EP) we have built for Staples for over 3 years, can be integration tested with much higher fidelity using simulation-based testing. Simulation tests combine the power of generative testing, explicit system state transitions, and system-wide integration testing. Less expensive than full-blown simulation testing is component-level generative testing using tools like Clojure's `test.check` to replace hand-written unit tests with dynamic and automatically generated tests. Unit tests and TDD still serve to refine component design and test very specific edge cases. You can watch Nivedita and Srihari discuss the EP architecture and simulation testing strategy in their Functional Conf 2015 talk, here: https://www.youtube.com/watch?v=YjfXhhxw9Bs

All of the above of course assumes not just a high level of automation but *complete* automation. To comprehensively test entire architectures with dynamically generated scenarios and interaction simulations, it must be possible to instrument and script the entire set of services. Often, this means that services are built "API first" to accomodate testing infrastructure in the way that object-oriented systems tended to be built "interface first" in the 90's and early 2000's when TDD and unit testing really took developer mindshare. Though

some projects we deliver continue to leave provisioning of hardware to the sysadmin team, absolutely everything else (orchestration, down) must be completely automated and scriptable or high-functioning testing strategies are impossible. The yin and yang in this situation is again reflective of bad modeling corrected by TDD and unit testing: If we try to write a test and get hung up on a manual process, the first thing we'll do is automate it.

See: "Scope of Work" 3. Develop and deploy the TM3 software See: "Scope of Work" 4. Developer and user documentation See: "Scope of Work" 5. Tests See: "How to Participate" 7. Testing (not user testing)

APIs

For over two years, the majority of nilenso's software delivery has been on systems with API front-ends. We are very familiar with growing APIs, managing API versions, defining "easy" APIs alongside "performant" APIs, and API-first development. We all appreciate a line of code is expensive to change, a database schema is much more expensive, and an API schema much, much more expensive than that. Particularly for APIs with unknown clients, "up-front design" is not a dirty term in any sensible agile team.

We see there are five major APIs pending in the existing TM2 software:

1. task details
2. validate/invalidate
3. lock/unlock a task
4. create task by position
5. create task by geography

None of these implementations looks particularly difficult and if we were to extend the TM2 codebase as-is, implementing draft versions of these APIs may be a good way to familiarize ourselves with the TM2 build/deploy process and overall design.

As mentioned above with regard to integration and simulation testing, it is often necessary in modern service architectures to design APIs "outside-in" for the sake of facilitating testing, even before the API has any clients.

In our experience, one major quality of well-designed APIs is a clear and explicit awareness of where and how state transitions occur. Objects and data structure instances are inherently state machines. Entire systems are no different, though it is extremely difficult (and inefficient) to make an entire system immutable, which means state transitions across systems are mutative and it is all the more important to capture state transitions for every API call which writes to a service.

It has only been implied until this point, but to make the point absolutely explicit: We will create APIs early with clear definitions, with the intention of

facilitating both testing and integration with other HOT software products.

See: “Goals” 5. API See: “Scope of Work” 5. Tests

Stack

We propose building TM3 in Clojure. We are not wedded to this idea.

The existing software (TM2) is built in Python and has serviced the community capably. However, we see a few issues with the existing architecture. After reviewing the Python/JavaScript TM2 codebase we found that there was no layering to the web architecture. All server-side components are built into “views”, which perform most of the domain logic and manage state transitions through user workflows. We certainly don’t insist that MVC, MVP, MVVM, or any other goofy acronym is the only way to build web applications or web services. However, at least one separation of server-side layers is often wise and TM2 would benefit from a more modular design. Because of the conflation of different responsibilities in the server-side view code, the test suite is quite cumbersome (though, thankfully, very thorough!). Reading the Python unit tests for TM2 do not give a clear sense of test intent and many even lack proper assertions. These problems can, of course, be addressed in Python by refactoring the existing codebase. However, we are not Python experts and we would venture a guess that it would take us longer to become fully proficient in the Python ecosystem than it would take us to rewrite the fairly thin 4000 lines of Python in TM2’s server-side code.

The incidental complexity of the Python code led us to run Radon over the codebase to get concrete metrics on cyclomatic complexity. 286 of the Python methods receive a grade of “A”, 15 receive a “B”, and only 2 receive a “C”. Granted, we would consider a “B” from Radon a failing grade. `osmtm/views/tasks.py#send_invalidation_message`, for example, receives a “B” with a CC score of 7, which we would consider very high. Still, the entire codebase is certainly amenable to refactoring and we are not recommending a rewrite in Clojure because the software is poorly designed.

We would leave the JavaScript as-is to begin with, though at 1000 lines of code, it’s likely that some of the JavaScript behaviour belongs on the server side as well.

Clojure is now a widespread and commonly accepted language. Though relatively new (at 7 years old) the development model is 60 years old and the Java/JVM ecosystem have been familiar to many for decades. Resources for learning Clojure are widely available and it’s now easy for anyone at any skill level to get started quickly. We recommend Clojure as it is our fastest language for delivery of software services and we have been building software in Clojure for over 6 years. Our entire team has primarily been doing Clojure development for over 2 years.

It is unlikely that the JVM is to be surprassed as a platform, virtual machine, or suite of GCs anytime soon. Millions of person-hours have hardened it as a platform. Clojure itself is also quite hardened. Since Clojure 1.3, very few significant language-level changes have occurred. With Clojure 1.8, we are now mostly seeing small performance enhancements and refinements to the language. Breaking changes rarely happen and are increasingly unlikely with future versions of the langugage. Although it's a much broader topic, Clojure's language design itself limits and discourages breaking changes in library interfaces, making a relatively new ecosystem surprisingly stable. Where the Clojure ecosystem is lacking, Java libraries can be consumed with no abstraction layer and no performance costs. REPL development is faster than Test-Driven Development and Design (though we tend to use both). A broad and mature approach to testing (including Simulant+Causatum for aforementioned simulation testing), proof, and documentation is standard in the Clojure community; a user base of LISP-come-JVM hackers or Java-come-LISP hackers means very few community members are distracted by shiny toys.

Thanks to the JVM, Clojure affords its users quite a few "free" advantages: monitoring, runtime inspection, profiling, performance tuning, swappable GCs, highly performant libraries, and a multitude of runtime environments. Monitoring, in particular, seems like a big issue for the Task Manager community… we noticed quite a few production issues even recently from the HOT Slack.

Python will of course be faster to start, since the very first commit will be an improvement and growth upon what has already been built. The OSM/HOT community appears to be very familiar with Python and that support could easily outweigh the many technical advantages of Clojure or our speed of delivery with it. We would be comfortable delivering this project on Python to continue refining the existing codebase if our proposal is otherwise attractive to the evaluation team. We are not a language-specific company and it is always our aim to build the best software we can for our customers – that doesn't always involve using our preferred technologies.

Regardless of language and stack, our infrastructure, automation, and deployment approach is always built abstractly on top of an OSS toolchain. Last year, for political reasons, we had to move all of our services for one client from AWS to a "private cloud". Had we built anything AWS-dependent, this process would have been impossible. Monitoring is essential. We run own our software in production from our first deployment, and we try to make that first deployment happen as early in our involvement as possible. Operations ("devops", these days) is a group effort but we are often even the first or second point of contact for services we build exclusively.

Unit, integration, functional, and acceptance testing is a part of the development process and a story isn't complete (or even deployable) until it is fully covered by automated tests. Simulation testing is a bit different and does not necessarily happen in lock step with development. User testing obviously cannot happen until a story is deployed to a testing or staging environment. This is generally

also true of cafe testing, to avoid having new users test software that hasn't been proved deployable on the true infrastructure.

See: "How to Participate" 5. Proposed software stack See: "How to Participate" 7. Testing See: "Evaluation of Proposals" 4. Proposed design and stack See: "Evaluation of Proposals" 6. Community engagement strategy See: "Make Shiny or Make New?"

Who is nilenso?

Permit us a brief recounting of our history.

Nilenso began over 3 years ago as a small web development shop. Our experience before starting nilenso was varied – and broad. Major languages included Java, Ruby, Python, C#, JavaScript, C/C++, Clojure, Haskell. Collectively, we had substantial contributions to major open source projects like The GIMP and GNOME. Our industry experience also spanned a wide range of verticals: from defense and policing to healthcare and climate change. Our initial projects were humbling; to begin with, we primarily built CRUD web applications for startups.

Very quickly, however, we moved away from web applications and began contributing to larger teams with more substantial architectures. Clojure became our language of choice, and our team began to include JVM hackers, PostgreSQL experts, and developers skilled in infrastructure automation. In our last year, we've grown both design and project management practices in addition to our core engineering team.

The software community in Bangalore found our story interesting enough to ask us to speak at Fifth Elephant, India's premiere big data conference: https://www.youtube.com/watch?v=b7K3E1Q_MBk

We have written about our most recent technical accomplishments:

http://nilenso.com/recent-tech.html

We will try to avoid repeating content from that document here, but we do have specific experience which is relevant to HOT, OSM, and the Task Manager Challenge.

Our Experience

We have built services and infrastructure within the Staples SparX machine learning team for over 3 years. Staples' Experimentation Platform (EP), in particular, is entirely designed, built, and operated by nilenso. EP handles many terrabytes of streaming data at a rate of 500 requests per second under a Service Level Agreement (SLA) of <10ms in the 99.9th percentile. Akshay

Gupta (@kitallis, on Twitter/IRC/Slack) currently leads the EP team and would be our tech lead for Task Manager 3.

We have presented on the Experimentation Platform architecture at Functional Conf 2015:

https://www.youtube.com/watch?v=YjfXhhxw9Bs

EP also involves nontrivial database clustering techniques, which have evolved with the lifespan of the project. Synchronous replication, failover to standby databases, ZFS snapshots, custom PostgreSQL MVCC AUTOVACUUM configurations, WAL streaming, realtime (non-star-schema) reporting replicas, db and I/O monitoring, partitioning, logical replication, load balancing, and multi-mode failure detection all must be understood by every member of our team working on database clusters.

We have presented on building a PostgreSQL db cluster at Rootconf 2016:

https://www.youtube.com/watch?v=sGJDg5ba0iI

We have contributed substantially to monitoring infrastructure in many recent software projects. This includes system-level, JVM-level, application-level, and db-level monitoring on all of our projects. We have been allowed to release some of the db and db monitoring work as open source. All of these repositorires are focused on PostgreSQL:

https://github.com/nilenso/honeysql-postgres (not monitoring specific) https://github.com/staples-sparx/Wonko https://github.com/staples-sparx/wonko-client https://github.com/staples-sparx/pg-cluster-setup https://github.com/staples-sparx/repmgr-to-zk https://github.com/nilenso/postgresql-monitoring

Unfortunately, although we have implemented internationalization (i18n) and built or extended translation tools on a number of projects, very little of this work is open source.

Similarly proprietary is our most recent legacy replacement work on the driver allocation service for Go-Jek Indonesia: We rewrote this service in Clojure with an extensive generative testing suite (and minimal unit tests for single-value edge cases). Thanks to generative testing and developer discipline, the service was released with zero production issues and completely replaced the legacy software on its first day in production. Ours was Go-Jek's first Clojure code.

This was not our first Clojure rewrite, however. We also rewrote Staples' configuration management, legacy data feed integration, and reporting tools in Clojure. This work involved a different style of testing. First, these services were built with large and comprehensive unit test suites for all known use cases. Then the new services were delivered alongside the legacy (Ruby) services in all environments and validated for multiple months. Once the Clojure services produced results consistent with the legacy services (as confirmed by automated validation software provided by us to the QA team), 3rd party services were redirected to new APIs and the legacy services were turned off. This process

was slow but not expensive and finished without a single interruption to existing clients.

We don't believe there is one perfect solution to writing correct software. We value strict type systems (like Haskell's), heavy external testing, test-driven development, generative testing, and mathematical proof differently based on the needs of each project. Correct, thoroughly tested software always makes project delivery faster, whatever the path taken.

Developers from nilenso have run training and courses in React, Haskell, and Clojure. We also speak regularly at conferences and the talks we haven't mentioned can be found here:

http://nilenso.com/talks.html

Our Future

The first few years of nilenso's existence have allowed us to walk the path from technical competence to technical excellence. We are very good.

But that isn't enough, is it?

Everyone wants to master their craft and we know this is a decades-long process. We all have a long way to go. But as each of us looks back on the work which brought us this far, the third axis becomes clearer and clearer, carved out against the first two axes of time and skill: meaning.

If mastery of a craft takes tens of thousands of focused hours, meaningful and impactful work takes lifetimes. We are all grateful for our position of privilege, given to us by uncountable predecessors, relentless and hardworking.

What does progress look like? How can we contribute positively?

We ask this question routinely, explicitly and implicitly, at nilenso. We know this is the goal we must move toward as a company and constantly evaluating exactly what it means to all of us is how we will find the way. For some of us, this work is the Arts. For some, international development. For others, childhood education. For some, Artificial Intelligence. As opportunities arise to contribute to these disciplines, we do our best to engage them. We have designed mobile user interfaces for India's rural hospitals with Samanvay. We have built citizen education platforms for fighting climate change with Fact0ry and mobile apps for detecting water quality metrics with Akvo.

We want to work with the Humanitarian OpenStreetMap Team.

Task Manager 3 is a wonderful opportunity. It is the first well-funded project we have had an opportunity to apply for in the mapping space. Free maps operate at a very high level in humanity's global progress: countless new tools can be built on OpenStreetMap. Individuals and software teams around the globe are completely unfettered when such systems and data exist. HOT pushes this

one step further by specifically targetting rapid improvement of maps for the areas which need them most. The Task Manager, by providing coordination and orchestration of teams, will bring efficiency to a global effort that would be otherwise impossible. TM3 is unlike many humanitarian software efforts we have previously evaluated, which don't involve a lot of data, infrastructure, or domain knowledge. Our experience makes us a good match for the management, design, and development challenges TM3 presents and we believe we will make this project a success.