Brian Conway
RCE Software, LLC
Version 1.0
June 28, 2018

Prisma Systems XSPACE: Moving To a Modular, Services-Based Architecture

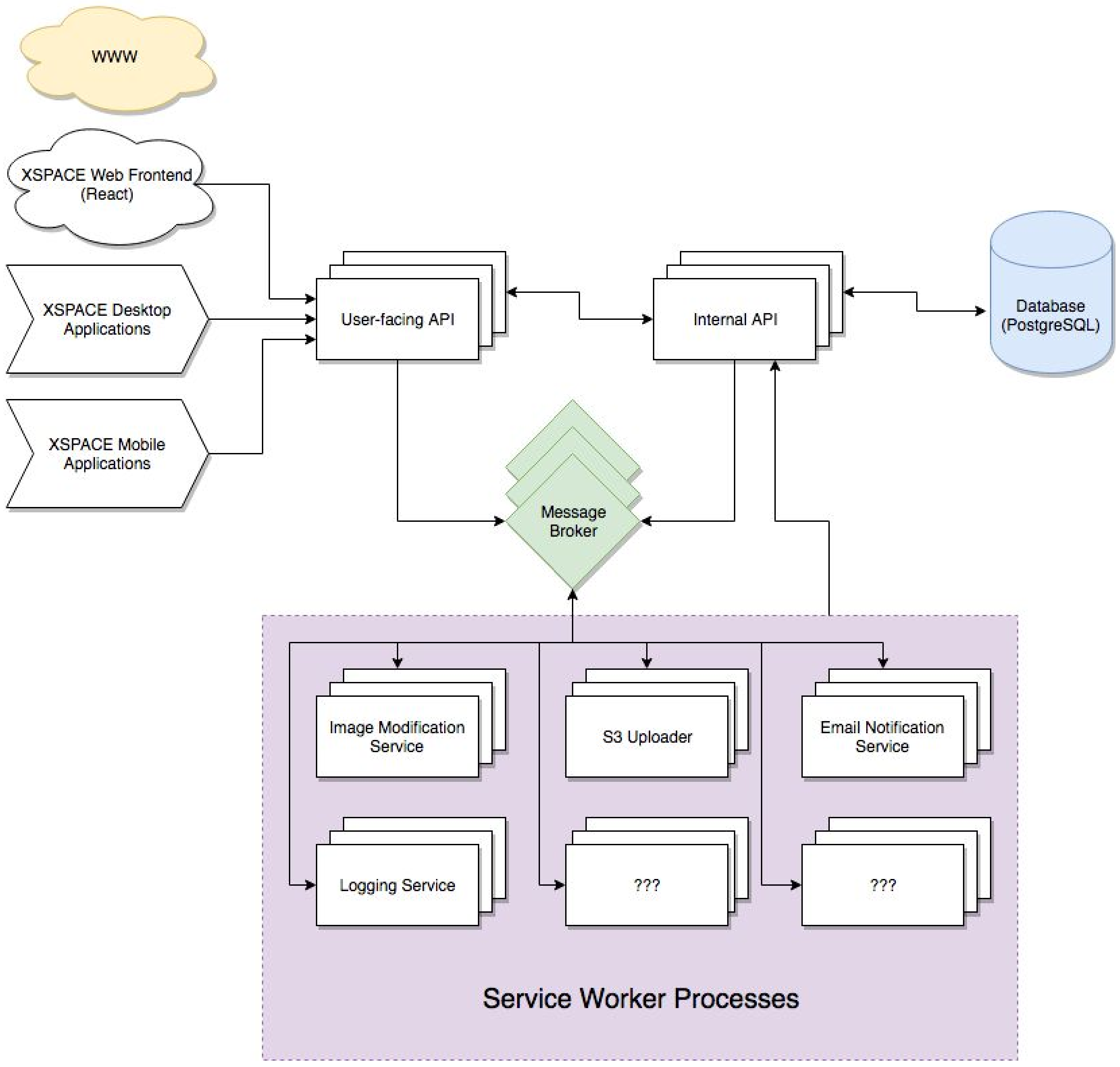
1.0 Introduction

With new customers signing up and going to production with the Prisma Systems XSPACE web application, a move from a monolithic Python stack to a modular, (micro)services-based architecture is needed to handle the expected growth and traffic by the platform. This document will outline an effective approach for designing and deploying such an architecture from the following perspectives: service layer and workflow architecture, deployment architecture design, future technology considerations, and load and performance tooling recommendations.

2.0 Service Layer and Workflow Architecture

The key to designing and building a scalable services-based architecture is: understanding the product's workflow, breaking that workflow down into unique tasks, understanding the resource requirements of those tasks (CPU, memory, and network), and scaling them horizontally. By understanding each unique task, we can visualize which are the "heavy lifters" and where our bottlenecks will form first. The architecture can also leverage asynchronous and concurrent processing, such that one process isn't impeding the work of others.

For example, an API call should not take a request from a client, update the database, upload a file, send an email notification, and then return the result to the user, all in a single set of serial actions. The more scalable approach would be to take the request from the client and return as quickly as possible, potentially updating the database first, and then running ancillary actions later and update the user on their results via a status.


WWW
XSPACE Web Frontend (React)
XSPACE Desktop Applications
XSPACE Mobile Applications
User-facing API
Internal API
Database (PostgreSQL)
Message Broker
Image Modification Service
S3 Uploader
Email Notification Service
Logging Service
???
???
Service Worker Processes

Service Layer and Workflow Architecture Diagram

2.1 Internal API

The primary role of the XSPACE Internal API will be to handle backend business tasks by manipulating the database and loading tasks into the message broker. The API's tasks will be short-lived and focus primarily on moving data from one location to the other - it will not do much heavy lifting. Scaling API web servers is trivial when coupled with shared database backend, using any number of available load balancing technologies, though it is unlikely more than a couple instances will be needed.

2.2 User-facing API

Like the Internal API, the User-facing API can be scaled horizontally. It will communicate with the user using a limited feature set compared to the internal API, and it will relay requests via the internal API rather than access the database directly. The User-facing API can also access the message broker directly to load new tasks, where applicable.

2.3 Database

PostgreSQL is a highly scalable database, and will scale well vertically with a single instance for even heavily-trafficked APIs. At the point where an infrastructure has maxed out even the larger Amazon AWS instances available, other options include high availability, load balancing, and replication. For the purpose of this architecture discussion, a single, well-provisioned PostgreSQL server will serve XSPACE well past the 100 customer mark.

2.4 Message Broker and Service Worker Processes

This portion of the architecture is where the bulk of the improved scalability is derived from. Lightweight messages are deposited into a high-performance message broker by either the Public API or Internal API, depending on the task. The choice of message broker platform should be based on routing and configuration requirements, durability, performance, and being a good match for technologies already in use by the stack. Most message brokers allow easy horizontal scaling, as well as scaling by locations. Popular examples include RabbitMQ, ActiveMQ, NSQ, and NATS.

From the message broker, the messages kick off any number of service worker processes that do the heavy lifting for the platform. These processes can scale near-infinitely as more instances are added. When complete, the service workers return their results via the Internal API. The examples shown in the diagram are not an exhaustive list, and they can include:

Image Modification Service: Any bulk image operations done after scanning will be CPU-intensive, and should be scaled outside of the APIs' fast paths.

S3 Uploader Service: Any task interacting with an external service over the network should be considered “slow” and avoid the API fast path. This is doubly the case when large file transfers are considered.

Email Notification Service: While emails are generally smaller in size than the images stored to S3, they are still subject to network effects, and may even include waits for retries, depending on the technology and packages in use.

Logging Service: Centralized, hosted logging services are convenient for both debugging and administrative notifications, but they are also best shelled out service process that won’t slow primary API transactions.

Countless more possibilities exist for parallelizing tasks in the XSPACE web application workflow as it matures.

Technical note: Care should be taken to design a retry plan for failed messages, either due to service worker failure or network failure. Rather than indefinitely retrying a failed message, setting a failed status and trying again with a mark-and-sweep technique can avoid resource avalanches.

2.5 XSPACE Web Frontend (React)

The React-based XSPACE web frontend is a client-side Single Page Application that can be deployed and served by a variety of methods: static hosting on S3, Node server installed on an instance, Heroku, etc. Regardless of which deployment method is chosen, the resources needed to serve this application are much lower than a server-side application and are trivial to scale with the appropriate load balancers. Once loaded, the JavaScript frontend communicates with the User-facing API.

2.6 XSPACE Desktop and Mobile Applications

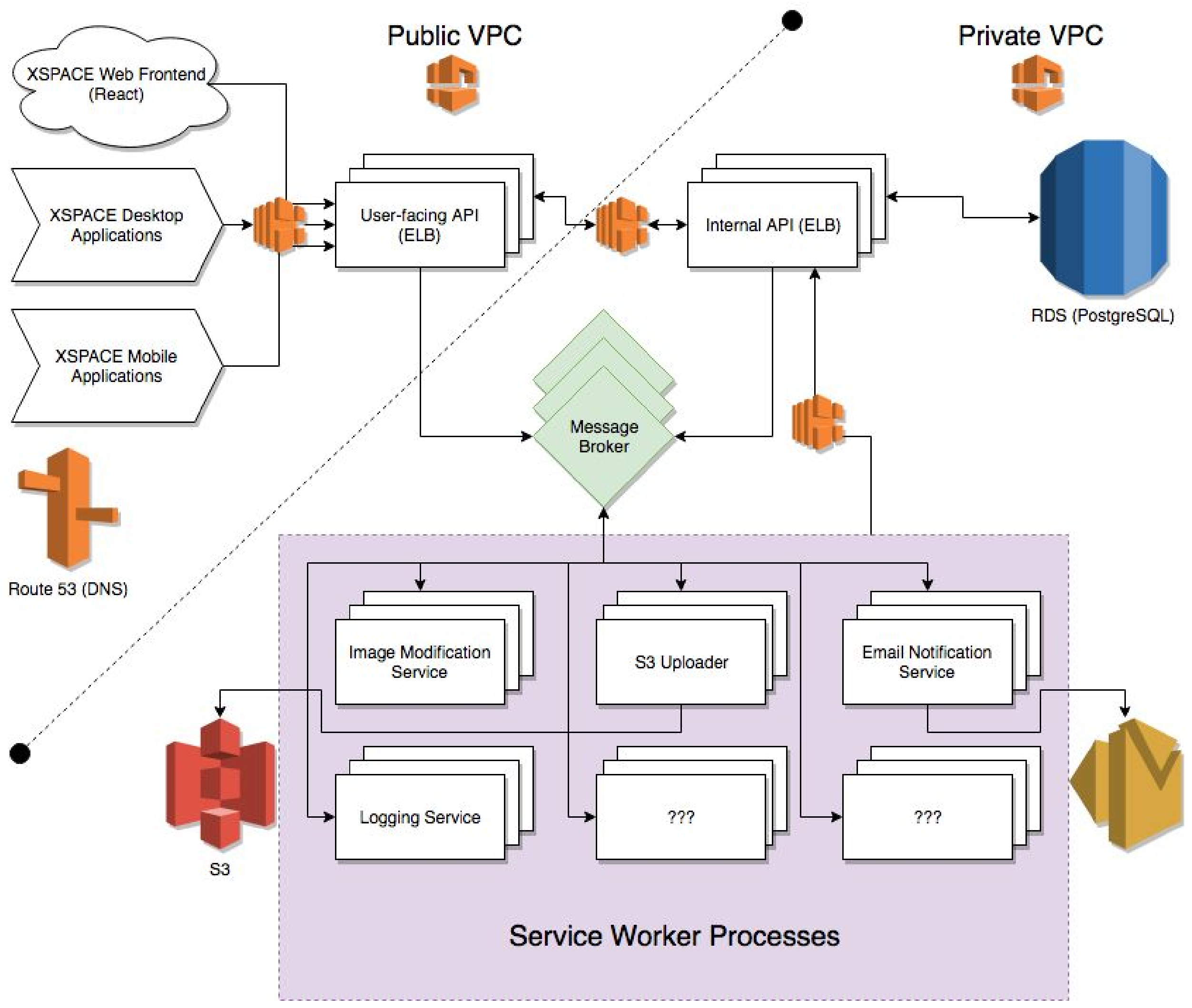
Similar to the React-based frontend, desktop and mobiles applications will also consume the User-facing API. As these apps are installed on client computers and devices, scaling them is not done as part of our infrastructure, and they will be taken into account with the scaling of the User-facing API.

2.7 WWW

The marketing/corporate web site is a static site that can be served scalably from any number of platforms. It is listed in the architecture for completeness, but is a standalone product.

3.0 Deployment Architecture Design

As part of the migration from a monolithic Python stack to a modular, services-based architecture, cloud-based computing and networking resources can be more efficiently leveraged than they are today. Amazon AWS provides a variety of managed product offerings that lower administration costs over traditional self-hosted software packages. Other cloud providers (Google, Microsoft) offering similar products, but this document will focus on Amazon AWS, where Prisma Systems is already engaged. Recommendations for both immediate implementation and future consideration follow.



Deployment Architecture Design Diagram

3.1 Amazon VPC

Amazon Virtual Private Cloud (VPC) allows users to build isolated network topographies via an easy web-based wizard. Using security groups and network ACLs, an administrator can configure their virtual network to provide public resources, while at the same time securing and limiting access to backend infrastructure. A multi-tier web application is a common use case for Amazon VPC, and is recommended here as illustrated above. A VPC is currently configured in AWS for the XSPACE Python stack, but it is not being used for isolation and all instances are currently public.
Recommendation: Required

3.2 S3

Amazon Simple Storage Service (S3) is a well-known object (data) store known for its availability, scalability, and durability. It is already in use as part of the XSPACE backend for 2D and 3D model storage. Recommendation: Already implemented

3.3 Route 53

Amazon Route 53 is high available and scalable cloud-based DNS. It is currently in use by Prisma Systems for a handful of DNS zones. Current usage is static assignment only, but in the future could be used to leverage dynamic host creation or advanced routing features. Recommendation: Already implemented

3.4 RDS

Amazon Relational Database Service (RDS) is a managed relational database supporting PostgreSQL, among other database engines. RDS significantly lightens the burden of database administration (though does not remove it completely), and supports easy database migration, version upgrades, and capacity scaling. It is currently used by the XSPACE Python stack today. Recommendation: Already implemented

3.5 ELB

Amazon Elastic Load Balancing (ELB) automatically distributes incoming application traffic across multiple targets, such as Amazon EC2 instances, containers, and IP addresses. ELB is not currently in use due to the single development stack deployed, but will be required under a services-based architecture as more frontend applications are deployed. Recommendation: Required

3.6 SQS

Amazon Simple Queue Service (SQS) is a managed message queue service that can replace a self-hosted message broker such as RabbitMQ, ActiveMQ, NSQ, and NATS. Two types of queues are offered based on ordering and durability needs. Message brokers are generally robust and painless services to administer, so SQS can be considered on a cost and administration need. It is important to note that moving to or away from SQS would both require code changes for the SQS API. Recommendation: Consider for the future

3.7 SES

Amazon Simple Email Service (SES) is a managed email sending service designed for transactional emails at low cost. It is one of many cloud-based email providers that do the hard work around keeping emails out of spam filters by applying all SMTP best practices. It is worthy of consideration, along with its competitors (Mailgun, SendGrid), for use by the email notification service workers. Recommendation: Consider for the future

3.8 Lambda

AWS Lambda allows companies to deploy code without managing servers. It is a newer style Function-as-a-Service offering (FaaS). Lambda can be an enticing offering, especially to replace smaller work processes with simple tasks, such as moving data between queues and S3. However, the usage costs should be carefully evaluated, as they can grow quickly compared to a fixed-rate EC2 instance on a frequently-used service. For a less-frequently run task, such as emailing a user, they may provide value. Recommendation: Consider for the future

4.0 Future Technology Considerations

With anticipated new customer sign-ups for Prisma's XSPACE web application on the horizon, it is important to be able to support rapid development and infrastructure growth with sensible language and technology choices. The considerations that follow will aid Prisma Systems in planning its technology roadmap.

4.1 Languages

When considering current and future programming language choices, the two most important factors are familiarity by the development team and technical sufficiency (maturity, correctness, performance). The Django Python stack implemented as the backend at the time of this document was a good choice for getting the Minimum Viable Product (MVP) off the ground. Python remains a suitable choice for both API (Django, Flask) and service worker process/microservices programming. Another consideration worth considering is the Go (Golang) programming language, pioneered at Google, and known for its rapid and typed development, safe and easy concurrency, performance, and extensive developer ecosystem. Go was created for building network services, and would mesh well with the direction of the XSPACE services architecture.

On the frontend, React remains a popular choice in the JavaScript ecosystem. Given the size and scope of the JS package world, maintaining the momentum with React and trying to minimize further dependencies would be advised.

Recommendation: Stay the course with Python and React, but consider alternatives at the services level as new services are added

4.2 Containerization

Containerization, most recently popularized by Docker, is extremely popular in the software development world. Two of the main advantages to containers include running software in isolation and building reproducible deployment artifacts.

While building containers with Docker can be a simple process, deploying and managing them is often less so. Amazon Elastic Container Service (ECS) is a container orchestration system, available through the familiar AWS console, for deploying and managing containers without additional server management. Moving to a more complex solution, Amazon also offers Amazon Elastic Container Service for Kubernetes (EKS). Kubernetes was written at Google, and offers a means to deploy, manage, and scale containerized applications. The administration overhead for Kubernetes can be quite high, as can the chance for tooling mistakes to cause downtime. It should be investigated for production use once the cost of traditional EC2 instance and container management grow too large for one team member.

Recommendation: Pursue reproducible deployment artifacts via containerization, but hold off on a complex orchestration system until needed

4.3 Secrets Management

Most of the security discussion thus far has focused on networks, firewalls, and API authorization. Often neglected in cloud infrastructures are database and server credentials. As an infrastructure grows and begins dynamically deploying services as the system load scales up or down, it is important to manage those credentials dynamically. HashiCorp's Vault is one such product that can dynamically update and enforce credentials across any service (database, API, etc).

Recommendation: Pursue automatic credential management when infrastructure credentials grow too numerous to manually manage

4.4 Service discovery and configuration

Continuing with the theme of scaling infrastructure dynamically, once a product has become popular enough to scale to many servers or instances, finding and coordinating these services becomes a problem. The currently accepted approach to coordination of dynamically deployed services is with network-based configuration and host management. Two popular tools/combinations for deploying this type of service are HashiCorp's Consul and a combination of Apache ZooKeeper and CoreOS's etcd. Of the two, Consul is more full-featured, while a combination of ZooKeeper and etc may provide value in the form of a customized stack.

Recommendation: Pursue automatic service discovery and configuration when the infrastructure grows too large to manually manage

5.0 Load and Performance Tooling Recommendations

As the Prisma XSPACE web application migrates from a monolithic Python stack to a modular, services-based architecture, it is important to keep a firm handle on the performance and bottlenecks of each service, such that they can be understood, monitored, and scaled individually. The following areas of architecture should be considered, benchmarked, and maintained: databases (PostgreSQL), REST APIs, service worker processes, and web servers. Many of these tools discussed below have paid SaaS offerings, but this document will focus on the tools themselves. Additionally, synthetic benchmarks will only give a user a glimpse of a service's behavior on a specific set of conditions. The best way to understand the performance of a service is to run production (or production-like) data through that system.

Note: This sections covers service testing and tuning, not development unit testing or integration testing.

5.1 PostgreSQL

pgbench is the de facto PostgreSQL benchmarking tool, and can be run against any self-hosted or RDS database. A couple good resources for understanding the results and tuning for them can be found here and here. It is important to consider and test with a database workload similar to that experienced in production.

5.2 REST API

Apache JMeter is one of the oldest and most-used tools for load testing a REST API. Its downside is that it requires having knowledge of your API endpoints and being a Java developer to program it. There are a number of JMeter SaaS offerings available, such as BlazeMeter and Loader.io.

Another option in the space of REST API testing is to write one yourself. Since a tester will need an intimate knowledge of the REST API endpoints being tested and their expected performance attributes, writing a customized testing tools in Python or another language of choice is not an uncommon occurrence. This is another scenario where the Go language could be considered for its scalability and performance.

5.3 Service Workers

When testing queue-driven service processes, the best way to drive them is by injecting tests directly into the message queue. Message brokers are generally very high performance and least likely to be a bottleneck in most architectures, so this is a fairly safe approach to driving service workers.

Service worker processes are another area where custom testing code is all but required. Knowledge of both the queue message format and the service worker output are needed to both verify success and calculate request rate. Python or the Go language would both hold up well for this task.

5.4 Web

ApacheBench (ab) is one of many popular tools for web server benchmarking. ApacheBench has been in use for a long time and features a single-threaded design that can become a bottleneck under extreme loads.

wrk is a modern HTTP benchmarking tool capable of generating significant load when run on a single multi-core CPU. It combines a multithreaded design with scalable event notification systems. Output is straightforward and easy to understand.

As noted above, web server benchmarking is also very popular in the SaaS space.

6.0 Conclusion

The move to a more scalable, services-based architecture will be necessary for the Prisma Systems XSPACE web application to handle the growth of upcoming customer onboarding and usage. This document outlined an effective approach for designing and deploying such an architecture from the perspectives of service layer and workflow architecture, deployment architecture design, future technology considerations, and load and performance tooling recommendations. This services-based architecture will easily last Prisma Systems through a number of years of growth before significant changes or re-architecture are required.