Foundation Phase Deliverables

Example Company | MVP

Completed on:	2023-04-26
Completed by:	Pavel Kirillov, Denis Stetskov

Checklist

✅ Identified technical challenges and added to the project kanban board.
✅ Decided and documented proposed architecture.
❌ Infrastructure deployed based upon proposed architecture - Phase 1 scope.
✅ Base projects for platforms setup - iOS, Android and Web boilerplates ready.
❌ CICD in place and operating correctly - Phase 1 scope.
❌ Modeled initial ERD of the proposed data models - Phase 1 scope.
✅ Modeled an initial infrastructure diagram of the system
❌ Documented API and provided link to doc - Phase 1 scope.
✅ Investigated third parties that will be used with examples.
✅ Running costs estimated for infrastructure and third party integrations. Ballpark.
✅ Investigated complex requirements and provided solutions with examples.
✅ Scheduled check up ceremony.

Architecture

Introduction

Example Company is a two-sided marketplace designed to connect investors with ventures needing funding, ultimately aiming to create a global private market.

Both founders and investors have pain around finding very tightly aligned opportunities and are seeking a matchmaker.

Example Company aims to reduce the friction and cost for both parties with their unique preemptive due diligence, matchmaking and deal room system.

Preemptive Due Diligence
All Companies closing investment rounds are required to go through due diligence. Example Company's preemptive due diligence process guides Companies through the due diligence process to increase the chances of companies attracting and securing Investors, while also de-risking opportunities for Investors. By doing the heavy lifting of due diligence ahead of the listing, Example Company speeds up the deal closing process.

Listing & Matchmaking
Investors are able to see listings of Companies that tightly align with their Investment preferences. Because of such thorough preemptive due diligence, when the Investor clicks to View Company Profile they are able to see great detail about the Company.

Deal Room & Closing Room
When an Investor and Company are mutually interested, they enter into a Deal Room to begin the negotiation process to close a Deal. The Deal Room provides a clear and customizable checklist to guide both parties to Closing. The checklist and documentation templates will be customized depending on the requirements of each Deal.

First Version of App Development Approach

We're going to learn to walk before we're going to learn to run a 10K race. Meaning that we will first focus on the happy case when devices are online, fast and powerful enough to do the work; users have decent stable and fast internet connection, devices have over 1GB of storage etc; web users are on most popular evergreen browsers like Chrome and Edge, have decent screen resolution and modern hardware, office networks aren't behaving like the Pentagon and connectivity is stable. Edge cases will be handled later. Cost optimization will become the focus only after the working end-to-end solution is developed and deployed.

We're going to assume that an average person will use the Example Company app between 2 and 45 minutes per day. Small numbers of Issuers and Investors will use the system significantly longer to set things up, enter the data and review deals.

Example Company doesn't fall into the high-load app category, but is expected to grow to thousands of daily users. Supporting thousands on the app shall suggest leveraging available Platform-as-a-Service vendors (AWS, GCP, Azure) to save time and focus on functionality and user experience vs dilemmas and experimentation to find the most cost-effective in-house approach.

The series of internal discussions with the senior members of the NineTwoThree development team have led to an outline of the initial approach to Example Company Architecture. During the discussions, we've reviewed several alternatives and formed a consensus that the app can follow our tried and true best practice template, which already addresses the common bottlenecks in this category of double sided marketplace apps.

Third-Party Components

A few third-party components we're going to use on the project are Google Firebase & Google Analytics. These two will help us monitor user behavior, ship Push Notifications, track App Crashes and hiccups, and maintain App stability.
We will use Prometheus and Grafana to monitor the infrastructure and send alerts.
Sentry.io will be our solution for server side error logging to help us debug backend issues.

Example Company considers both the Issuer and the Investor data and documents to be critical information. Data leakage and loss incidents are likely to damage the reputation of the business beyond repair, therefore posing an existential risk to the business. During the foundation, we had a series of meetings with advanced security vendors in search of the best in class solution used by the large players in the industry such as banks, insurance and financial institutions. We will review the TransmitSecurity solution for identity management, verification and authentication further in the document, and explore Alloys' solution for the "bad actor check" process required in the main flow of the application as well.

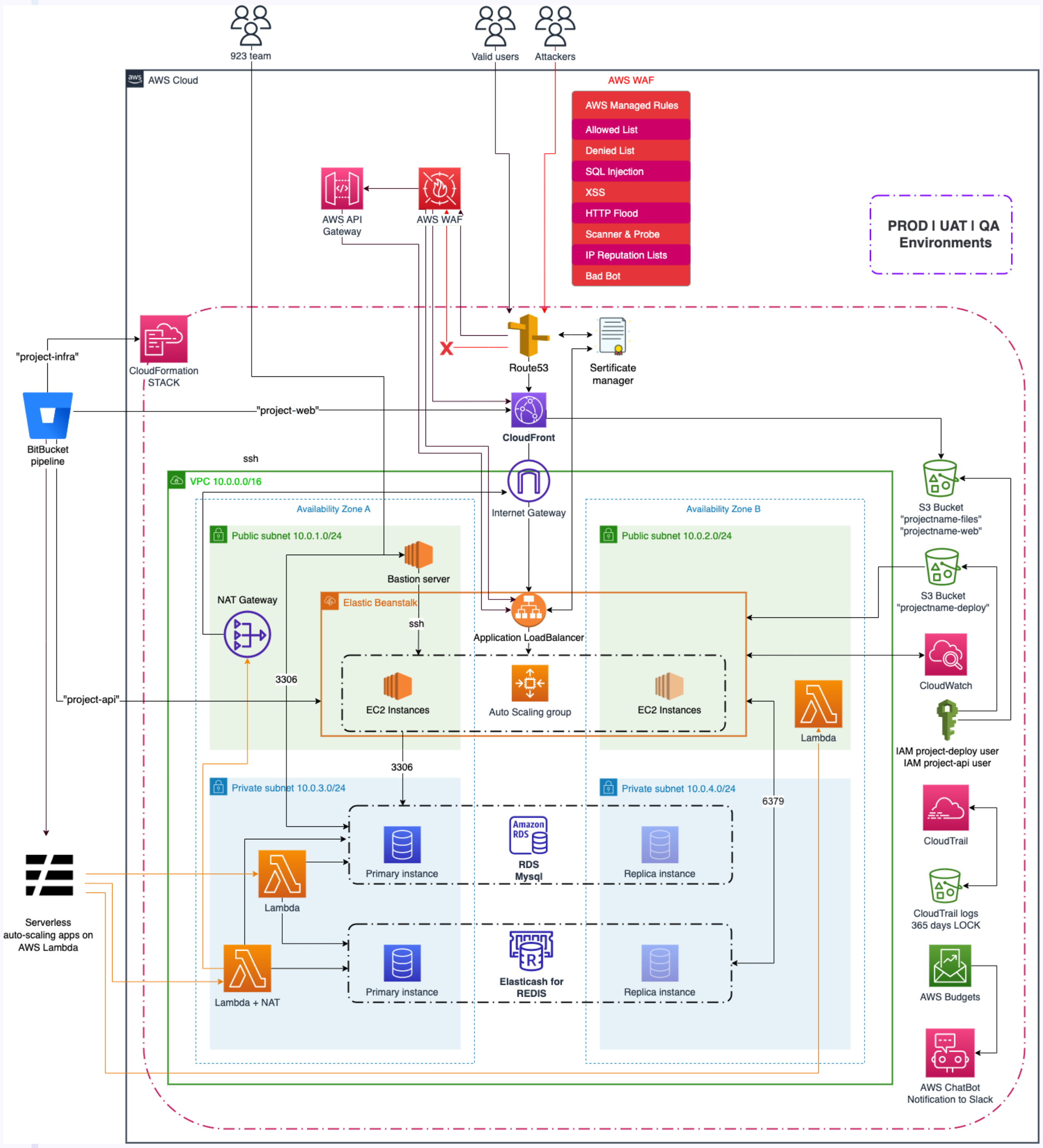
During the Foundation, Example Company has indicated their interest in eSignature and Notarization solutions, including the option to build it in-house. We have explored the various vendors in the industry, who have developed their eSignature and Notary solutions and would recommend DocuSign as the best in class vendor offering aggressive pricing that matches offers made by their competition. To save the upfront development costs (which we estimate in extra months of effort) and reduce uncertainty risks, we'll recommend implementing the eSignatures & RON using DocuSign and their Notary API. In future phases, after the product is stable and proven by the Example Company, we'd recommend doing the cost / benefit analysis of developing and replacing DocuSign functionality by developing the in-house solution to cut costs paid to DocuSign. Our team does not see the obvious value for developing this in-house in the near future.

Example Company shall consult with their legal team regarding the feasibility of using the Remote Online Notarization (RON) in the app flow, as it is not authorized universally across US and around the world. For example, as of the time of this writing we have found evidence online that California, Delaware, Connecticut and Massachusetts (just a few examples) do not accept RON. There were temporary permits introduced to address COVID-19 pandemic restrictions that have expired by now.

We recommend Example Company selecting AWS as the PaaS (Cloud) Vendor. Amazon have pioneered this market and remains the market leader today. NineTwoThree is a Certified Partner of AWS. We have multiple certified Solution Architects, as well as a full DevOps team in-house.

nine
two
three

High Level Diagram



Infrastructure

Description

Diagram above describes our current vision or the system. Let's discuss the round trip of data on a very high level.

The infrastructure set up by CloudFormation for the system consists of several AWS services that work together to create a scalable and reliable environment. The services used include Route 53, CloudFront, ELB, RDS, and AWS Lambda, all of which are orchestrated by CloudFormation. The deployment process is managed using Bitbucket pipelines, which provides a streamlined, automated deployment pipeline.

Route 53 is used for domain name registration and DNS routing, providing a reliable and scalable way to manage traffic to the system. CloudFront is a CDN (content delivery network), distributing content to users from multiple edge locations around the world, reducing latency and improving the user experience.

ELB (Elastic Load Balancer) is used to distribute incoming traffic across multiple instances of the system, ensuring that traffic is evenly distributed and that no single instance becomes overloaded. RDS (Relational Database Service) provides a scalable and reliable database service, with automated backups and failover capabilities, ensuring data is always available.

Lambda functions are used to run crons or background tasks, which can be triggered by time, API Gateway, or other events. This allows for efficient processing of tasks without the need for maintaining separate servers. The Lambda functions are deployed using the Bitbucket pipeline, which automates the testing, building, and deployment process, ensuring that changes are deployed consistently and reliably.

By using CloudFormation to orchestrate these services, the infrastructure can be managed and maintained in a consistent and repeatable manner, reducing the risk of errors or inconsistencies. This approach also allows for easy scaling and management of the environment, as well as enabling the use of infrastructure-as-code practices, making it easier to automate, version control and manage changes to the infrastructure.

In addition to the services mentioned above, the system also uses AWS WAF (Web Application Firewall) and API Gateway to enhance security and manage API requests. AWS WAF is a firewall that provides an additional layer of security by inspecting incoming web traffic and blocking any malicious requests, protecting against common web exploits such as SQL injection and cross-site scripting. WAF rules can be customized to meet the specific needs of the system, providing a flexible and customizable way to secure web applications.

API Gateway is used to manage and control API requests, providing a scalable and secure way to expose the system's functionality to external applications. It allows for fine-grained control over API access, including authentication and authorization, rate limiting, and caching, ensuring that only authorized and authenticated requests are processed, and that API resources are used efficiently. API Gateway can also be used to transform and optimize incoming requests, enabling the system to process data in a more efficient and streamlined manner.

Overall, by utilizing services like WAF and API Gateway, the system can be made more secure and efficient, while also providing a flexible and scalable way to manage and control API requests.

The concise explanation of the key features and services used does not include unnecessary details. All the core components of the architecture were mentioned above.

Security Considerations

All communication will happen over a secure socket layer of HTTPS. All requests over HTTP will be automatically redirected to HTTPS.

The Bastion server will be set up to enter the Monitoring and Logging data. Access to Databases will only be possible through SSH tunneling. Production and Development environments will be separated.

We have not included AWS WAF, AWS Shield and other services that will protect us against DDoS and XSS threats to keep the high level diagram easy to understand.

One potential solution for Example Company to consider is a combination of Transmit Security and Alloy for bad actor checks. Transmit Security is a unified identity and access management platform that can help secure login and authentication processes. Alloy, on the other hand, is a declarative language used for modeling software systems and analyzing their properties.

The combination of Transmit Security and Alloy could provide a comprehensive solution for identifying and preventing malicious activity, as well as improving the efficiency of security operations. In the following sections of the document, we will describe each of these solutions in more detail and explore how they could be integrated to meet the specific needs of Example Company.

Scalability, Availability & Redundancy

MySQL will run as a cluster. We have weighed our options to use MySQL or PostgreSQL. MySQL Suitable for applications with high volume of reads. PostgreSQL Suitable for applications with high volume of both reads and writes.
CloudFront will serve static files from S3 Origins.

Backup & Disaster Recovery

Full DB backups will be created every 12 hours. We will store two snapshots per day for 14 days. Then 1 snapshot per day for the next 14 days then weekly snapshots made on Sunday nights for 6 months or longer if necessary.

Code base will be hosted in a managed Git repository. The NineTwoThree uses Bitbucket. Deployment will be automated with CI/CD scripts. The backups are managed by the vendor.

Project Setup

Repositories

Source code of the apps is hosted on NineTwoThree BitBucket folder

Android repository will be added once we start.

iOS repository will be added once we start.

Backend repository will be added once we start.

Frontend repository will be added once we start.

Continuous Integration & Continuous Delivery

Both Frontend and Backend CI/CD will be configured on Bitbucket Pipelines.

Monitoring, Logging & Exception Reporting

Exception Reporting will be configured during the development phase.

Prometheus, Grafana, ELB will monitor the main system KPIs. CPU, Memory, Storage, Network and additional ones that will help us fine-tune the load balancing and scaling. The Infrastructure Team will determine additional KPIs.

Additional attention should be allocated to AWS Billing alerts. We will create AWS billing alerts to avoid unexpectedly getting an astronomical bill. The alerts are configured by our DevOps team before we start deploying any infrastructure.

Shared Slack channel for alerts shall be configured to deliver alerts to both Example Company and NineTwoThree teams.

The NineTwoThree Internal Tools

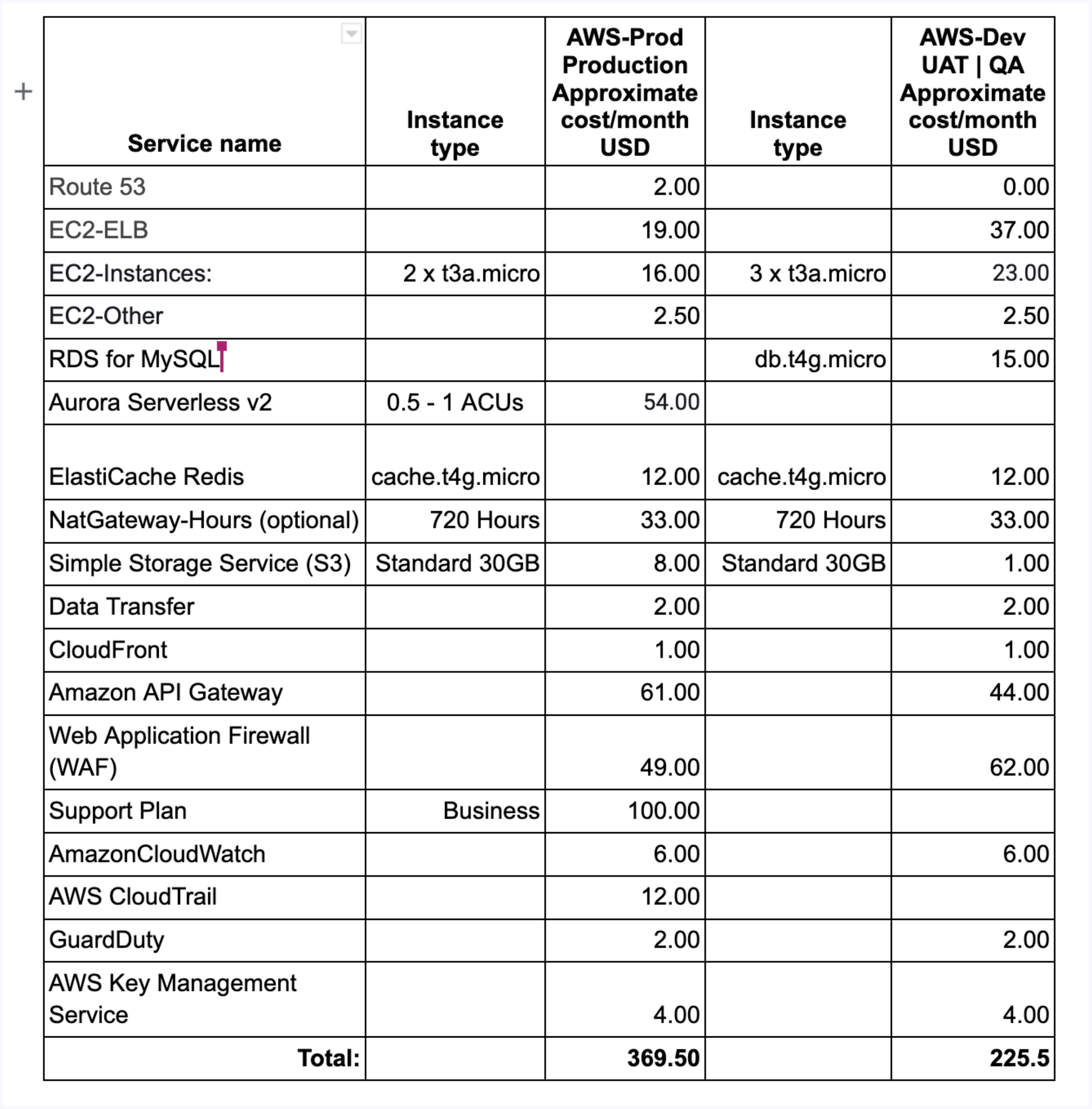
The following internal tools will be used to cover the corresponding requirement:

- Project Management : Monday.com
- Issue Tracking : Monday.com
- Customer Action Item Tracking : Monday.com
- Code Repository: BitBucket
- Internal Communication: Slack

Estimated Costs

Infrastructure Costs

The estimated initial infrastructure cost for the system is $630 per month. However, it is important to note that this cost may vary depending on the level of user adoption and usage of the system. The actual cost depends on factors such as the amount of traffic the system receives, the size of the user base, and the resources required to process and store data. Therefore, while we expect the initial cost to be around $630 per month, it is possible that the cost may be higher depending on the system's usage patterns. Then it is expected to scale linearly. In estimation, we include next Amazon Services:



Service name	Instance type	AWS-Prod Production Approximate cost/month USD	Instance type	AWS-Dev UAT | QA Approximate cost/month USD
Route 53		2.00		0.00
EC2-ELB		19.00		37.00
EC2-Instances:	2 x t3a.micro	16.00	3 x t3a.micro	23.00
EC2-Other		2.50		2.50
RDS for MySQL			db.t4g.micro	15.00
Aurora Serverless v2	0.5 - 1 ACUs	54.00		
ElastiCache Redis	cache.t4g.micro	12.00	cache.t4g.micro	12.00
NatGateway-Hours (optional)	720 Hours	33.00	720 Hours	33.00
Simple Storage Service (S3)	Standard 30GB	8.00	Standard 30GB	1.00
Data Transfer		2.00		2.00
CloudFront		1.00		1.00
Amazon API Gateway		61.00		44.00
Web Application Firewall (WAF)		49.00		62.00
Support Plan	Business	100.00		
AmazonCloudWatch		6.00		6.00
AWS CloudTrail		12.00		
GuardDuty		2.00		2.00
AWS Key Management Service		4.00		4.00
Total:		**369.50**		**225.5**

In addition to AWS - the paid service we would like to recommend using is Postmark App, a mailer we're using on most of our projects. Cost is $10 for 10000 emails and goes down from there. We're expecting the Example Company project to remain under $100 tier for the first year.

For messaging and chat functionality, we suggest getstream.io as the best-in-breed vendor. Initial plan will cost 500$ per month and cover all functionality needs of the app.

It has an SDK for all platforms. Example Company should try to get accepted to their startup program, which allows free use in the beginning.

In the future, there will be a dilemma - to continue paying getstream.io for their solution or to re-implement everything with an in-house solution. Right now, we strongly recommend buying. Stream.io has solved the problem of scalability and ease of implementation. Another advantage is that we're using stream.io on multiple projects for the last couple of years and are familiar with the SDKs and API. This will allow us to deliver the initial scope predictably fast and grow into a decent scale with it. Then when the bill will reach 5 digits - cost optimization will make sense.

For quick response on issues, we will use Sentry.io which will notify us about any production errors. It should cost ~30$ per month.

The PowerHouse team shall negotiate other third party contracts, such as: DocuSign, TransmitSecurity, Alloy, etc. These vendors have custom pricing and special plans for startups. We have not included their pricing in this document.

Extended Review of TransmitSecurity solution for Example Company

TransmitSecurity provides a passwordless authentication solution that eliminates the need for traditional username/password combinations. Instead of entering a password, users are asked to complete an action, such as confirming an email or clicking a link sent to their phone.
Here is an example of implementing passwordless authentication using TransmitSecurity:

1. Start the authentication flow: The client requests the server to initiate the passwordless authentication flow.
2. Send a notification: The server sends a notification to the user's email address or phone number.
3. Complete the authentication action: The user receives the notification and clicks the link or confirms the email, completing the authentication action.
4. Verify the authentication action: The server verifies that the authentication action was completed successfully.

5. Generate an authentication token: The server generates an authentication token for the user and sends it back to the client.
6. Authenticate the user: The client uses the authentication token to authenticate the user.

With TransmitSecurity, the implementation details of the authentication flow are handled for you, allowing you to focus on integrating passwordless authentication into your application. To get started, you'll need to sign up for a TransmitSecurity account and obtain an API key. You can then use the API key to initialize the TransmitSecurity SDK in your application and use its API to implement the passwordless authentication flow.

Simple example and description of implementation we provide bellow:

JavaScript
```javascript
const transmit = require("transmit-security");

transmit.init({
  apiKey: "your-api-key"
});

// Step 1: Start the authentication flow
app.post("/start-auth-flow", function (req, res) {
  const userIdentifier = req.body.userIdentifier;

  transmit.startAuthFlow({
    userIdentifier: userIdentifier,
    channel: "email"
  }).then(function (response) {
    res.send({ message: "Authentication flow started 
successfully." });
  }).catch(function (error) {
    res.status(500).send({ error: error.message });
  });
});
```

```
// Step 4: Verify the authentication action
app.post("/verify-auth-action", function (req, res) {
  const userIdentifier = req.body.userIdentifier;
  const authActionToken = req.body.authActionToken;

  transmit.verifyAuthAction({
    userIdentifier: userIdentifier,
    authActionToken: authActionToken
  }).then(function (response) {
    res.send({ message: "Authentication action verified
successfully." });
  }).catch(function (error) {
    res.status(500).send({ error: error.message });
  });
});

// Step 5: Generate an authentication token
app.post("/generate-auth-token", function (req, res) {
  const userIdentifier = req.body.userIdentifier;

  transmit.generateAuthToken({
    userIdentifier: userIdentifier
  }).then(function (response) {
    res.send({
      message: "Authentication token generated
successfully.",
      authToken: response.authToken
    });
  }).catch(function (error) {
    res.status(500).send({ error: error.message });
  });
});
```

In this example, we use the transmit-security library to call the init method with our API key. Then, we implement the passwordless authentication flow as a series of RESTful API endpoints. The startAuthFlow method initiates the authentication flow and sends a notification to the user. The verifyAuthAction method is called to verify that the authentication action was completed successfully. Finally, the generateAuthToken method generates an authentication token for the user.

The level of effort required to implement a general authentication flow and a passwordless authentication flow using TransmitSecurity can vary depending on many factors, such as the complexity of your application and the level of customization required. However, here are some general observations:

General authentication flow: Implementing a general authentication flow from scratch can be a complex and time-consuming process. You need to create the necessary user accounts, manage password storage and encryption, implement email verification, and handle forgotten password scenarios, among other things.

Passwordless authentication flow with TransmitSecurity: Implementing a passwordless authentication flow using TransmitSecurity can be much simpler, as TransmitSecurity provides the necessary infrastructure and security measures to handle the authentication process. All you need to do is integrate the TransmitSecurity SDK into your application and use its API to initiate the authentication flow, verify the authentication action, and generate an authentication token. This can be accomplished with a few lines of code and is typically much faster and easier to implement than building a general authentication flow from scratch.

In conclusion, using TransmitSecurity for passwordless authentication can greatly reduce the effort required to implement an authentication system, especially if you are looking for a secure, fast, and efficient way to authenticate users in your application.

For Example Company, using TransmitSecurity for passwordless authentication offers several benefits, such as enhanced security, improved user experience, increased efficiency, and compliance with industry standards. However, there are also potential disadvantages to consider, such as dependence on a third-party solution, required infrastructure, and potential limitations on customization.

To successfully implement TransmitSecurity, some assumptions need to be made. For example, users will need to have access to a smartphone or biometric sensor in order to complete the authentication process, and a reliable internet connection will be necessary for communication with the TransmitSecurity API. Additionally, integration with existing systems, adherence to appropriate security protocols, and the use of industry-standard security practices will be essential for a successful implementation.

By considering these factors, Example Company can make an informed decision about whether using TransmitSecurity for passwordless authentication is the right choice for their security needs, and can take steps to ensure that the implementation is successful and effective.

Extended Review of Alloy solution for Bad Actor Check

The implementation of an alloy bad actor check in Node.js can vary greatly depending on the specific requirements and goals of the system, but some common methods include:

- IP reputation checks: Checking the reputation of an IP address to determine if it is associated with known malicious activities.
- User behavior analysis: Monitoring the behavior of a user over time to determine if it deviates from normal patterns and potentially indicates malicious intent.
- Input validation: Checking user input for signs of malicious intent, such as SQL injection or cross-site scripting (XSS) attacks.
- Rate limiting: Limiting the number of requests a user can make in a given time period to prevent excessive usage and potential abuse.
- Honeypots: Setting up traps for malicious actors to detect and identify them before they can cause harm.

It is important to note that no single method can provide complete protection against all types of malicious actors, and a robust security system will typically employ a combination of these methods to ensure the highest level of protection.

In general, implementing a basic bad actor check in Node.js is relatively straightforward and can typically be done within a few hours or days, depending on the experience of the implementation team and the complexity of the system. Implementing a more comprehensive and secure bad actor check will likely require a greater effort, potentially taking a few weeks or months to complete, depending on the requirements and constraints of the system.

To accurately assess the effort required for implementing Alloy for bad actor checks, Example Company should provide clear requirements and goals that they intend to achieve with the use of Alloy. This will help to establish a baseline understanding of the project scope and identify the necessary resources, time, and effort required to effectively implement Alloy for bad actor checks.

Simple example and description of implementation we provide bellow:

JavaScript
```javascript
const Koa = require('koa');
const rateLimit = require("koa-ratelimit");
const Router = require('koa-router');

const app = new Koa();
const router = new Router();

// Set up rate limiting to prevent excessive requests
const limiter = rateLimit({
  db: new Map(),
  max: 100, // limit each IP to 100 requests per windowMs
  duration: 15 * 60 * 1000 // 15 minutes
});

// Apply rate limiting to all requests
app.use(limiter);

// Input validation to prevent malicious payloads
```

```
app.use(async (ctx, next) => {
  ctx.request.body = ctx.request.body || {};
  await next();
});

// Honeypot to detect and prevent malicious bots
router.get('/hidden', (ctx) => {
  ctx.status = 404;
  ctx.body = 'Page not found';
});

// Route for normal requests
router.get('/', (ctx) => {
  ctx.body = 'Welcome to the application';
});

app.use(router.routes());
```

In this example, we are using the koa-ratelimit library to set up rate limiting and prevent excessive requests, and the koa-router library to handle routing and input validation. The rate limiting will limit each IP address to 100 requests every 15 minutes, and if a user exceeds this limit they will receive a 429 Too Many Requests response.
This is just a basic example, and in a real-world implementation, we may need to add more security measures and customize the implementation to meet your specific needs.

A honeypot is a security tool that is used to detect and deflect malicious activity. It works by setting up a fake target that appears to be an attractive target for attackers but is actually being monitored by security personnel.

When an attacker interacts with the honeypot, security personnel can use this interaction to gain insight into the attacker's methods and motivations and take appropriate measures to protect the real target. This can include implementing stronger security controls, patching vulnerabilities, or taking legal action against the attacker.
There are different types of honeypots, including:

- Low-interaction honeypots: These are simple honeypots that simulate only a small part of a system, such as a single service or protocol. They are typically easy to set up and maintain, but may not be as effective as high-interaction honeypots.
- High-interaction honeypots: These are more complex honeypots that simulate a full system, and allow attackers to interact with the system in a more realistic manner. They are more resource-intensive, but provide a more accurate view of attacker behavior and can provide a greater level of protection.
- Honeynets: A honeynet is a network of honeypots that are deployed together to simulate a complex network environment. This allows security personnel to observe multiple types of attacks and to better understand attacker behavior.

Honeypots can be a valuable tool for enhancing the security of a system, as they can provide early warning of attacks and can give security personnel the information they need to respond quickly and effectively. However, honeypots should be used with other security measures and should not be relied upon as the sole means of protection.

For Example Company, implementing Alloy for bad actor checks offers several benefits, such as improved security through the prevention and detection of malicious activity, early warning of potential attacks, better understanding of attacker behavior, and increased efficiency through automation of certain security tasks. However, there are also some potential disadvantages to consider, such as the resource-intensive nature of Alloy implementation, the possibility of false positives, and the need to use Alloy in conjunction with other security measures to provide adequate protection.

In order for Example Company to successfully implement Alloy for bad actor checks, some assumptions need to be made. For example, the implementation will require technical expertise, particularly for high-interaction honeypots, which can be complex to set up and maintain. Additionally, regular maintenance will be necessary to ensure Alloy is functioning effectively and keeping up with changes in the threat landscape. By considering these factors, Example Company can make an informed decision about whether implementing Alloy for bad actor checks is the right choice for their security needs.