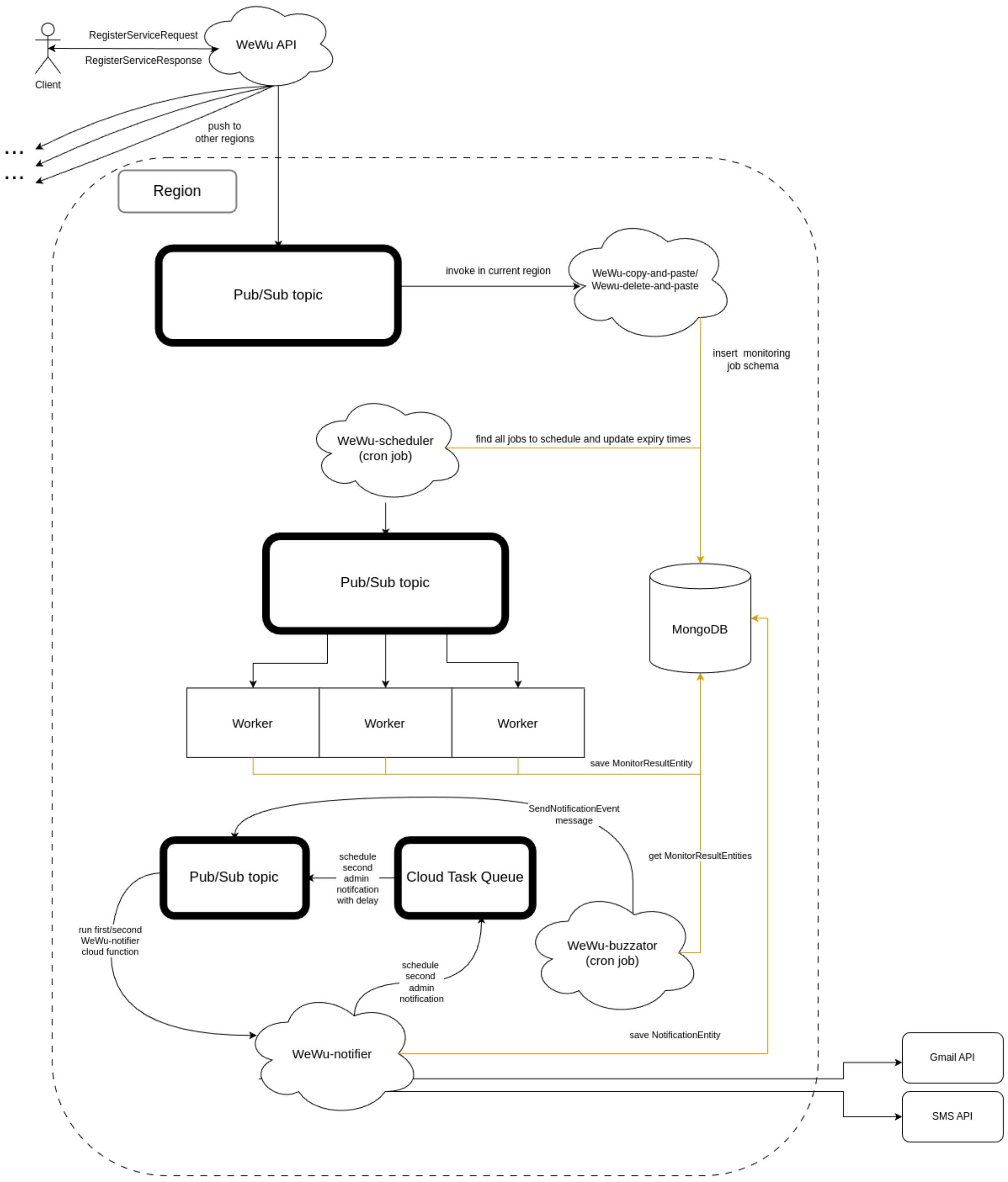
WeWu alerting platform

Design Schema



Client
RegisterServiceRequest
RegisterServiceResponse
WeWu API
push to other regions
...
...
Region
Pub/Sub topic
invoke in current region
WeWu-copy-and-paste/ Wewu-delete-and-paste
insert monitoring job schema
WeWu-scheduler (cron job)
find all jobs to schedule and update expiry times
Pub/Sub topic
MongoDB
Worker
Worker
Worker
save MonitorResultEntity
SendNotificationEvent message
get MonitorResultEntities
Pub/Sub topic
schedule second admin notifcation with delay
Cloud Task Queue
WeWu-buzzator (cron job)
run first/second WeWu-notifier cloud function
schedule second admin notification
save NotificationEntity
WeWu-notifier
Gmail API
SMS API

Overall design

Each cloud function which is not scheduled by cron job will be triggered by events. Each event will be represented by a Pub/Sub topic, and each topic will have its Avro schema.

WeWu-API (cloud functions)

User sends a request to the API informing that we should register a service to be monitored.

POST {wewu-api-register-cloud-function-url}

```java
class RegisterServiceRequest {
        serviceUrl: URL,
        geoRegions: List<GeoRegion>,
        primaryAdmin: ServiceAdmin,
        secondaryAdmin: ServiceAdmin,
        pollFrequencySecs: Integer,
        alertingWindowNumberOfCalls: Integer,
        alertingWindowCallsFailCount: Integer,
        ackTimeout: Integer,
}
```

If in *alertingWindowNumberOfCalls* consecutive pings service won't respond with correct status code *alertingWindowCallsFailCount* times, then the alerts will be sent to the admins accordingly. *ServiceAdmin* and *GeoRegion* are defined as follows:

```java
class ServiceAdmin {
        email: Optional<String>,
        phoneNumber: Optional<String>,
}

enum class GeoRegion {
        US_EAST("us-east1-b"),
        EU_CENTRAL("europe-central2-a"),
        ASIA_SOUTHEAST("asia-southeast1-a")
}
```

Either *email* or *phoneNumber* is required. Although *ServiceAdmin* could contain many ways of communication, we restrict it to have exactly one way - this approach is more implementation-friendly and it's conforming to the specification. If we were provided with invalid *email* or *phoneNumber*, then 400 is returned.

WeWu-API at *RegisterServiceRequest* generates random UUID, and pushes *CopyPasteInatorPayload* onto the *scheduleJobEvent's* Pub/Sub. The payload is a register request, but decorated with the random UUID - in order to identify the same job in different geographical locations. The same message is sent to all regions' topics specified in the request in order to trigger the Cloud Functions which will be responsible for further processing (saving to the database to be precise).

DELETE {wewu-api-delete-cloud-function-url}?jobId={jobId}

Jobs can be deleted from the database by making the above request, by delete operation we mean marking the job as canceled in the database. Where the *jobId* is UUID taken from *RegisterServiceResponse*. **WeWu-API** will publish the message in all supported regions to delete the job with the received *jobId*. **Wewu-delete-and-paste-inator** will handle this task further.

WeWu-API support Cloud Functions

WeWu-copy-and-paste-inator (cloud function)

This function will be responsible for:

- Saving job into the database (it takes CopyPasteIntorPayload)
- ~~Propagation of tasks between regions, in case the user wants us to monitor his service in multiple regions~~. We want to lift responsibility of inter-continental traffic from WeWu-API, because the API is supposed to be quick and we would like it to respond to the user with no time taken (inter-continental transfer will have its latency). Being an event-driven cloud function allows us to retry it in case of failure and to invoke it asynchronously.

```java
class CopyPasteInatorPayload : RegisterServiceRequest {
      jobId: UUID // This will be the same request as registering,
                  // but decorated also with jobId
}
```

Scheduling the job in current regions means saving the job to the database - the **WeWu-scheduler** will take care of distributing work between workers. The Job database schema is defined as follows (*expirationTimestamp* informs us when we should reschedule the job):

```java
class JobSchema {
      jobId: UUID,
      serviceUrl: URL,
      primaryAdmin: ServiceAdmin,
      secondaryAdmin: ServiceAdmin,
      pollFrequencySecs: Integer,
      alertingWindowNumberOfCalls: Integer,
      alertingWindowCallsFailCount: Integer,
      ackTimeout: Integer,
      isCancelled: Boolean,
      expirationTimestamp: Long
}
```

WeWu-delete-and-paste-inator (cloud function)

This function will be completely analogical to the one described above, however it's only responsibility will be to change the *isCancelled* flag to *true*.

What's our database engine?

For the database engine we chose MongoDB (deployed using Atlas - automatically, it provides us with connection url, which is then propagated to the all parts of the system by environment variable) because it is optimized for storing large collections of small documents, also it provides simple transactions and database operations which is useful when it comes to the notifications processing - this is the main reason, because we would like to for example sort things and aggregate them. We also considered plain old good PostgreSQL, but despite being powerful, it lacks an ability to scale, which would impact our service significantly. We also considered a Firebase based solution, but MongoDB rich documentation and previous experience with this database drove us to choose this solution as Firebase DB is a part of a more complex system and we would like to be able to mock the DB easily in the integration tests.

WeWu-scheduler (cloud function)

This service is responsible for pushing tasks onto the monitor tasks queue. We define a queue task as monitoring some service per *timeQuantSecs* (maximal time of processing job by the **WeWu-Worker**). **WeWu-scheduler** will run full database search every 20s (this value is a subject to change) and retrieve all rows, where expiration date has passed, also in the same query, it will update expiration date to be *currentTimeSecs* + *timeQuantSecs*. This would be a http-invoked cloud function, invocations will be made by the Cloud Scheduler (cron job).

```java
class WorkerMonitorTask {
        jobId: UUID,
        serviceUrl: URL,
        pollFrequencySecs: Integer,
        taskDeadlineTimestampSecs: Integer,
}
```

All these fetched tasks will be put onto the Pub/Sub to be consumed by the **WeWu-worker**s. This will be implemented as a Cloud Function scheduled as a cron job. If needed, this component can be scaled by sharding the database manually and giving the responsibility of pushing tasks onto the queue to the multiple Cloud Function instances (1 database shard = 1 function). Although sharding in our implementation would be performed manually, we will ensure that this can be semi-manual process - we define a metric for cloud function invocation length and we can set alerts if this invocation time will exceed some limit, then we will know that scaling is required for further expansion of this system.

Design Tradeoffs

Having cloud function with semi-manual autoscaling being responsible for this task has some drawbacks:

- We can't scale automatically, although we can set custom metric for "need to scale" alerting
- We are increasing the usage of the cloud functions quota, but only a little bit

What are the advantages of this situation?

- We can use terraform to quickly deploy our solution (in this project we use one template for every cloud function and adding another function entity equals to 5 lines of new code in deployment schema)
- We can write quickly our solution, which is especially needed considering the deadline we have (we don't need to worry about server configuration)
- Algorithm for sharding and determining whether notification should be sent when it comes to specific job is relatively simple, which highly increases our ability to deliver MVP before the deadline
- To implement cloud functions we can use Python which significantly shortens the time needed to develop the application. It is possible thanks to our previous experience with this language in the Cloud environment as well as the huge variety of libraries available.

Why are we using Pub/Sub?

Because it offers task queue services and allows us to automatically distribute tasks between **WeWu-worker**s and also, it provides us with a metric we could use for auto scaling purposes (how many unfinished tasks are there in the queue). We would like to have a topic for each geographical region to separate them from each other. This also enables us to avoid trans-continental network transfers when it comes to the task distribution. Of course, some minor trans-continental transfer still occurs because of the initial task scheduling and deleting it from the monitoring system - we can't avoid this overhead because tasks for monitoring from different regions need to be scheduled. This doesn't really have competitors, because using a database as "Pub/Sub lookalike" wouldn't scale that efficiently and wouldn't provide us with automatic metrics, which will be used in **WeWu-worker** autoscaling.

WeWu-worker (pods managed by k8s Deployment) [repository]

One worker can process up to 20 (we don't commit to this number as it's probably too low, and it will be configurable) tasks. Processing job is described as follows:

1. Message is fetched from the queue which is local (in the sense of region) and its lease is set to *timeQuantSecs*. However, Pub/Sub allows modifying the ACK deadline only up to 10 minutes, so if the *timeQuantSecs* is longer than that, we will extend the deadline a few times to match the deadline specified in *WorkerMonitorTask*.
2. Worker will parse the information contained in the Pub/Sub message and will proceed to monitor service according to the received parameters. Message structure is defined above (*WorkerMonitorTask*).
3. Workers will monitor the service until the *taskDeadlineTimestampSecs* has passed and then it will ACK it - message expiration time is set by the **WeWu-scheduler**.
4. Worker instance will be saving ping info data to 2 tables in MongoDB:

a. First table will be configured to utilize Time To Live feature (it serves as a fast lookup source), there will be 8h of data in this database.
b. Second table will be configured as long time information storage, it will store all information about pings done in the past (without Time To Live feature) - (it will be time series). We plan to archive all of the pings, because they may be used in "data science" in the future (e.g. displaying charts for the end user how their service performs over time).

Database schema of the monitoring result is defined as follows (one ping):

```java
class MonitorResultEntity {
        id: UUID,
        jobId: UUID,          // standard desc index
        timestamp: Long,
        expiresAt: Date,      // time to live index
        result: MonitorResult
}

enum class MonitorResult {
        SUCCESS("SUC"),       // 2xx or 3xx
        FAILURE("FAIL"),      // 4xx or 5xx
        ERROR_TIMEOUT("ERR_TO"),
        ERROR_DNS("ERR_DN"),
        ERROR_NO_RESPONSE("ERR_NR");
}
```

How will we provide high availability?

The **WeWu-worker**s are going to be deployed using GKE regional clusters in Autopilot mode in order to reduce platform administration overhead and provide nodes high availability, auto-scalability and auto-healing. To distribute the workload across the regions we will create a cluster in each region and register all the clusters to the fleet, ~~and configure Multi Cluster Ingress to route the incoming traffic.~~ Worker service will have its Deployment specified so the pods could self-heal in case of any failure. When a service has a chance to perform a graceful shutdown (i.e. the SIGTERM was sent), all the tasks assigned to the pod are going to be released so the other workers could start processing them (so basically that means the worker will nack all the messages). If a graceful shutdown couldn't be performed for some reason, the messages would go back to the queue in *worker.ackDeadlineSecs=600* (worst case scenario). Therefore, we allow for situations where services will not be monitored for this period of time, which is certainly a drawback that we have taken into account. However, this value is configurable and can be lowered if you allow for more frequent ACK deadline modifications. Additionally, the deployment will have *maxUnavailable* and *maxSurge* set in the way to eliminate application unavailability during updates (didn't choose particular values yet). To ensure the scalability of the pods we will have HPA specified according to our needs. The **WeWu-worker** will scale based on an external metric from PubSub**.** Additionally, it will have its VPA declared to resolve the issue where some workers are overloaded with high-frequency pinging while the rest would have only lightweight jobs.

WeWu-buzzator (cloud function)

This will be implemented as a Cloud Function scheduled as a cron job. If needed, this component can be scaled by sharding the first ("small") database and giving the responsibility of scanning the database to the multiple Cloud Function instances (1 database shard = 1 function). Function will be executed every 20 seconds and it will be scanning a "lighter" MongoDB table (the one with only 8h data) using aggregation (in Python code) in order to detect a service failure. After failure detection *SendNotificationEvent* will be sent to the **WeWu-notifier** by pushing a message onto the Pub/Sub.

Design Tradeoff

As in **Wewu-scheduler** we will have the same drawbacks and advantages: fast deploy and fast development, but semi-manual scaling. There will be also one more cost (related to the database engine selection):

- We will have to perform aggregation in our source code (joins are non-optimal when it comes to the NO-SQL database like MongoDB). This part will be covered in Python, but it can be rewritten to some compiled language if necessary (however, a probable slow part would be transferring data to and back from the database).
- This function will have to store a large part of documents in memory, so its costs could be potentially higher than those of other instances.
- After consideration, aggregation could be implemented in the relational database, but it would require a complex and nested window function and table join (query would be very expensive and difficult to monitor). We've implemented it in Python, we know that it's slower than performing it on-prem (in the DB instance), but code maintenance and testing is way easier and that's what we'd like to have in a fast-paced startup environment.

WeWu-notifier (cloud functions)

Why is WeWu-notifier a cloud function(s)?

We would make the most of Cloud Task Queue which enables us to schedule a cloud function invocation in the future. What is more, event-driven cloud function comes with exponential backoff which we will utilize in case that notification can't be delivered to the admin.

First cloud function:

Notifier is an event-driven Cloud Function with exponential retries and retry windows which firstly gets executed by **WeWu-buzzator** (not directly, but with Pub/Sub) in case notification should be sent. The execution flow looks like this:

1. Cloud function receives the payload *SendNotificationEvent*.
2. Insert into MongoDB information about notification (*NotificationEntity* below).
3. Send the mail to the primary Admin through the API.
4. Save audit log to the database. We decided to use the database here as the data should be persistent.

5. Adds task to Cloud Task Queue which will publish the message to the Pub/Sub in the future.

Second cloud function:

Cloud task queue is set to start (by pushing to Pub/Sub) **WeWu-notifier** second cloud function execution. Execution originates from the Cloud Task Queue and it pushes message onto the Pub/Sub (to get Event Driven cloud function to automatically retry failed runs):

1. Coud function receives the payload *SendSecondNotificationEvent*.
2. Finds the *NotificationEntity* by *jobId*, if the *ACKed* field is set to true – cloud function ends.
3. If not, we send the notification to the secondary Admin.

```java
class NotificationEntity {
	notificationId: UUID,
	jobId: UUID,
	primaryAdmin: ServiceAdmin,
	secondaryAdmin: ServiceAdmin,
	ackTimeoutSecs: Integer,
	acked: Boolean,
}
```

```java
class SendNotificationEvent {
	notificationId: UUID,
	jobId: UUID,
	primaryAdmin: ServiceAdmin,
	secondaryAdmin: ServiceAdmin,
	ackTimeoutSecs: Integer,
}
```

```java
class SendSecondNotificationEvent {
	notificationId: UUID,
}
```

Notifier tradeoffs

Although using Cloud Tasks Queue allows us to schedule publishing message to Pub/Sub, it also creates another complication in application design. Additionally we cannot use Cloud Tasks Queue alone because HTTP Cloud Functions do not support retrying. Using Gmail API also comes with limitations as it has a send mails limit, but for our purposes it should not cause any problems.

ACK-ing the notification

Notification ACK-ing will be implemented as HTTP invoked Cloud Function - user will be provided with address to invoke function in the mail message.
https://<cloud function_url>?notificationId=2137548

Logging

For exploring the logs we will use Cloud Logging. It allows for easy searching, sorting, and analyzing logs. Helps with detecting errors and gathering logs regarding specific errors. It has a highly configurable retention period (between 1 day and 3650 days). GKE automatically collects the logs generated by non-system containers, so there is no overhead configuration needed to benefit from it. Same applies for Cloud Function logs.

Alerting

For alerting we will configure notifications in the Error Reporting page to report liveness probe failures in order to know our own system health status. Also, we will be using alerts to detect unacceptable execution times for **WeWu-notifier** and **WeWu-buzzator**.

Metrics

We would like to measure the following events:

- Every minute we report the number of tasks processed by the worker.
- Every minute we report the number of pings done by the worker.

Everything will be integrated with Cloud Monitoring.

Testing

All services will run automated unit and integration tests as part of continuous integration (CI) to validate their sole responsibilities. For instance, the tests for **WeWu-worker** will ensure it correctly handles new tasks, pings the necessary services, and maintains the appropriate polling frequency etc. For integration tests, we will use Pub/Sub emulators and Testcontainers to closely mimic the real environment, minimizing the use of mocks wherever possible. To conduct end-to-end (e2e), load, and stress tests, the testing environment will be set up on GCP with fake service. Its responses are going to be configurable through an API (very simple one, like /response/500 would change all its responses to 500 until the next change in the configuration).

Test cases:

1. Monitoring services
    a. Configure the alerting platform to monitor a set of known HTTP services.
    b. Intentionally make one of the services unavailable.
    c. Check if the alerting platform detects the service unavailability (possibly by checking the audit logs, for simplicity)

2. Notification sent and escalation process to the secondary admin
    a. Simulate a service failure.
    b. Verify that the alerting platform sends a notification to the primary system administrator.
    c. Check if the notification contains precise information about the service failure.
    d. Wait for the *ackTimeoutSecs* and expect a second notification to be sent (or not to be sent, if you want to check if ACK works).
3. Load test
    a. Start with a baseline load of tasks (e.g. 200)
    b. Gradually increase the load by adding additional concurrent tasks in predefined increments (e.g. 15 tasks every 6 seconds).
    c. Continuously monitor system performance metrics, including response time, throughput, and resource utilization during the load increase.
4. Stress test
    a. Simulate a large number (let's say *x*) of tasks with high poll frequency.
    b. Monitor resource usage such as CPU, memory – VPA should increase the resources available. Additionally, expect the number of pods to increase to *x / worker.maxTasksPerPod (*configurable in worker service, but remember to change in manifest if needed).
    c. Generate a high volume of simultaneous service failures.
    d. Monitor the alerting platform's response time and accuracy under stress.

CI/CD

For continuous integration/deployment we will build pipelines on Github Actions.

Pods workflow:

- Build Maven Package
    - Build Jar
    - Run unit tests (with Surefire Plugin)
    - Run integration tests (with Failsafe Plugin, using Testcontainers for Pub/Sub emulator and MongoDB, because we want the environment to be the closest possible to the real app)
    - Build and push Docker image to the Docker Hub
- Deploy the application to GKE (possibly for each region)
    - Deploy the Deployment manifest
    - Deploy the VPA manifest
    - Deploy the HPA manifest

Functions workflow:

- Build and Push Docker image
- Check file formatting (isort, black)
- Run unit tests
- Run integration tests
- Deploy Terraform (on deploy tag)
- Destroy Terraform (on destroy tag)