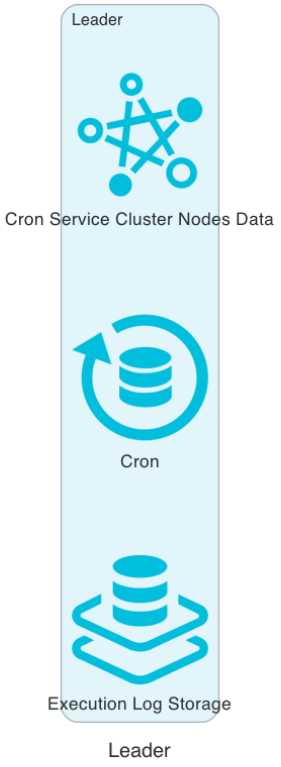
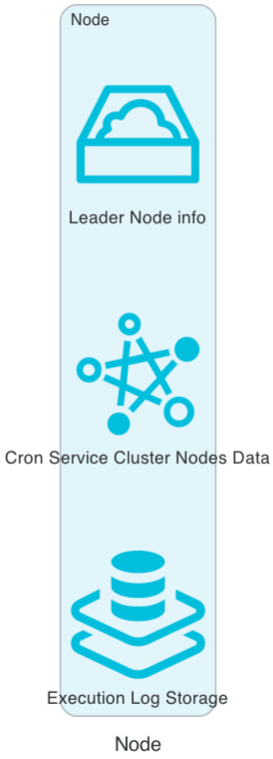
Design a Distributed Task Scheduler

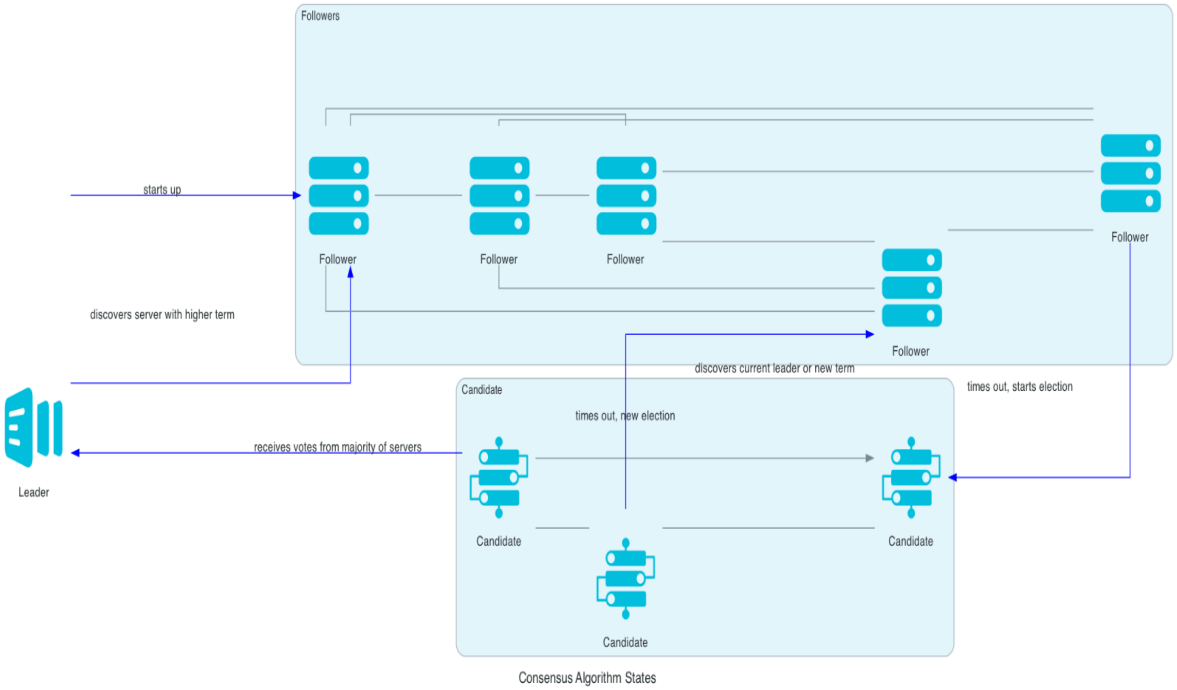
I can see few different approaches for this:

1) "Practical" - The simplest and cheapest approach is to use a tool that solves this problem, for example, DKron or similar solutions. Once we understand what doesn't fit our use cases, we can look for alternatives or develop our own solution.
2) "Classical" - As I know, most big tech companies use something similar to this: Distributed cluster(s) of computer nodes that use consensus algorithms like Paxos or Raft.
    - Leader node: Maintains the cron schedule and information about the cluster, passes execution of jobs to follower nodes.
    - Follower nodes: Maintain information about the cron schedule, jobs, and leader node, and keep eventual backups of cluster data. As shown in the following diagrams:



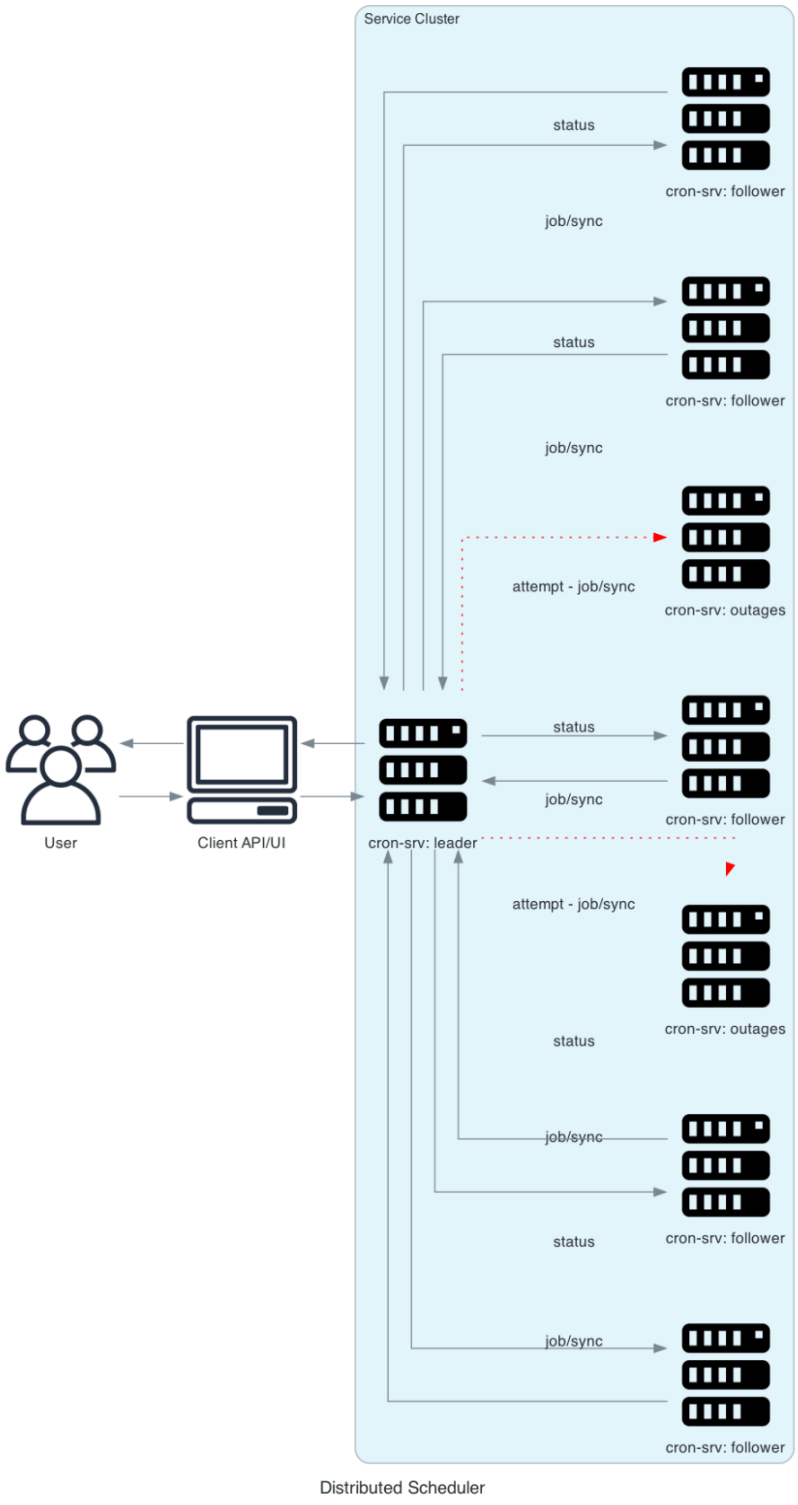


And to elect a new leader, for example in Raft, they use something similar to:



Consensus Algorithm States

Here how End Users can interact with this scheduling clusters:



This approach usually means low-level development, custom protocols, and careful support 😕. It's a pretty interesting engineering task, but in my opinion, not every company really needs this. Usually, we already have some tool that can help us build a scalable system.

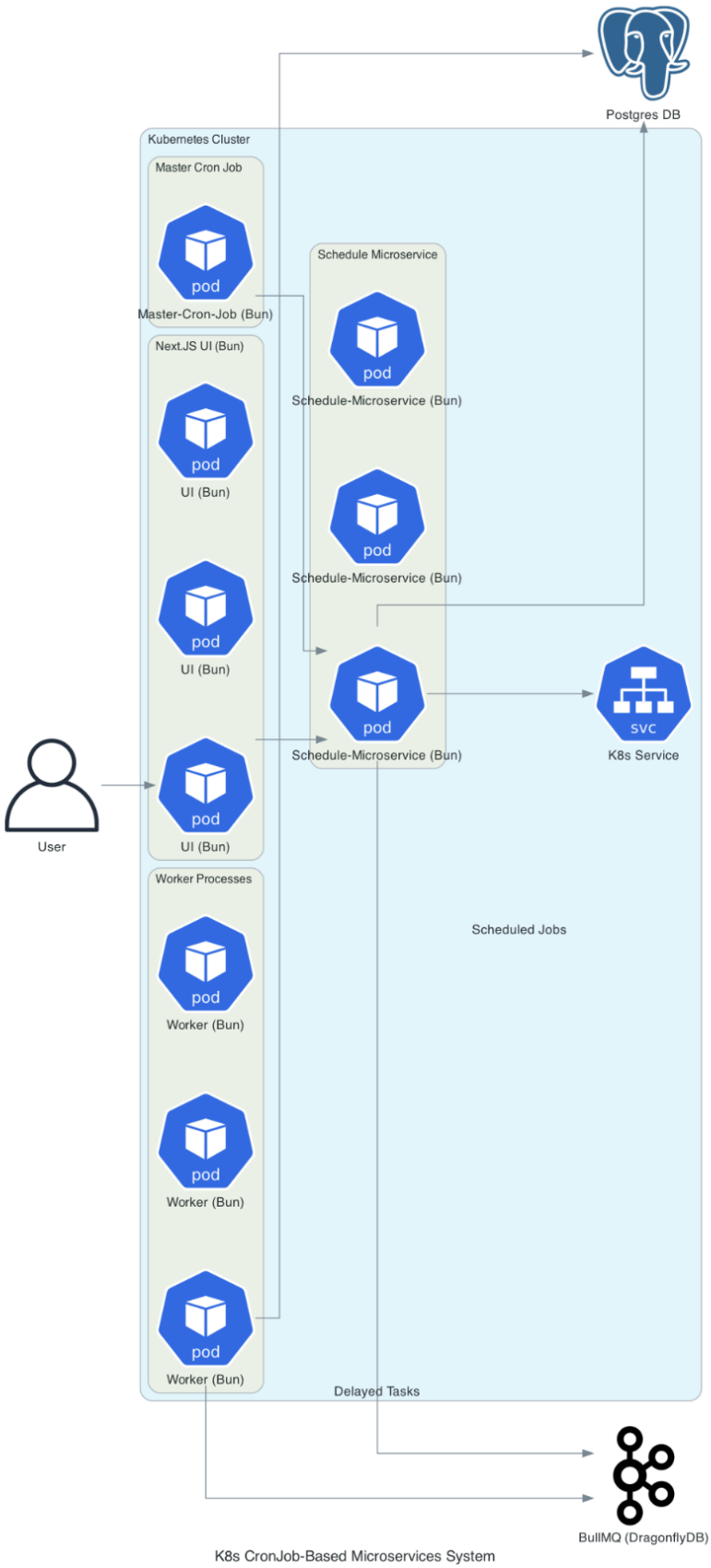
3) Using modern tools and their capabilities (which we usually already have) like K8s, message queues, databases, etc.

   During this exercise, I released a workable prototype of this approach.

   Steps:

   - K8s has a CronJob service that can run the app in a pod. If the app fails, K8s takes care of it.
   - K8s runs a simple application at a specified interval, for example, every 30 minutes or 10 minutes, etc.
   - This app can divide the interval into one or more smaller chunks and pass each chunk to another app - the schedule-microservice - to look into jobs in the specified intervals and plan their execution.
   - The “schedule-microservice” is behind a K8s service (acting as a basic load balancer) and has multiple replicas.
   - “Plan executions” means calculating when a job needs to be executed and putting it into the message queue as a delayed task (there may be more than one execution during an early specified interval).
   - “Worker” processes (multiple) maintain the queue and execute tasks. They can use pub/sub to distribute events about execution.

   UI - a separate app, uses the “schedule-microservice” to maintain scheduled tasks



K8s CronJob-Based Microservices System

During the implementation, I decided to make the process more enjoyable by experimenting with several new tools and technologies:

- Code Editor: I planned to try Zed (https://zed.dev/) and even started using it a few times. While it was nice to use, it felt too "raw" with some visible and annoying issues, a lack of plugins, and an inability to debug apps.
- Bun (https://bun.sh/) instead of Node.js: This provided the best development experience. Bun solved CJS/ESM issues, supported native TypeScript, and was super fast. It also offered pretty good support for Node.js features. However, it lacked mocking and had some issues with unit tests. While it's possible to use third-party tools like Jest/Vitest, Bun's integrated tools weren't as fun to use.
- TS app to TS app with gRPC/Protobuf: This was a nice experience, similar to working with these technologies in other languages. But types from DB to UI and back.
- Next.js 14 with the new App Router: The lack of documentation was initially confusing, but it felt similar to the recently announced React v19.
- BullMQ: (https://bullmq.io/) It just worked. It was pretty simple to use, and I chose it for its support of delayed jobs and deduplication.
- Diagrams as a code (https://diagrams.mingrammer.com/), you can found sources and generated in distributed-scheduler/diagrams
- Turborepo, (https://turbo.build/repo) helped mitigate issues when no CI/CD process and available artifacts/package storage

Cons of implementation:

- Potential Overengineering: It might seem like overengineering for what was asked, but I enjoyed the process of coding it myself. This approach allows the system to scale up or down in specific areas.
- Lack of Unit Testing: There was a lack of unit testing due to issues with Bun's testing and mocking capabilities, as well as a lack of time.
- Code Duplication: There is some code duplication in different projects (type mapping). I'm considering refactoring or moving this to a library project, or using tools like Automapper.
- Possible defect, but this is prototype
- Definitely naming not ideal

Key points of implementation:

- Implemented Design: The suggested design was fully implemented in code with all components, including a simple user service.
- Durability and Flexibility: The system is durable and flexible, thanks to Kubernetes and the message queue. It is also cost-effective compared to the described "classical" approach.
- Types Throughout All Layers: Types are used consistently across all layers, from the database with Prisma to gRPC/Protobuf, to microservices, applications, and the UI.
- UI Features
    - Design Approach: Tried not to follow the "our design is programmer" approach.
    - Mixed Server/Client Side: Implemented a mix of server-side and client-side rendering for optimal performance.
    - Loading Speed: Ensured fast loading times.
    - Dark Mode: Automatic dark mode based on the operating system settings.
    - Data Validation and Error Messages: Implemented thorough data validation and clear error messages.

Some UI overview (no sound):
https://www.loom.com/share/4af138859f7c446d8924adb110a1daf5?sid=918f6965-a7f7-4e10-b28a-11bcd4fcad86

Run / Deploy:

1) Locally

Requirements:

- Bun (https://bun.sh/)
- Postgres
- Redis-like db

DBs can be run in docker:

Redis-like, I used DragonflyDB (https://www.dragonflydb.io/)

```
docker run -p 6379:6379 --ulimit memlock=-1
docker.dragonflydb.io/dragonflydb/dragonfly --cluster_mode=emulated
--lock_on_hashtags
```

Postgres:

```
docker run --name local-postgres -e POSTGRES_PASSWORD=password -e
POSTGRES_DB=app_ds_db -e POSTGRES_USER=db_user -p 5432:5432 postgres
```

In distributed-scheduler@schedule-apps

```
bun install
```

Make DB:
In: distributed-scheduler@schedule-apps/packages/db

```
bun run prisma
```

In distributed-scheduler@schedule-spps

```
Bun run build
bun run dev
```

UI should run on http://localhost:3000
* Check log, if port not available Next.JS can change it

Only master-cron-job app require periodically / as needed manual run
In distributed-scheduler@schedule-repo/services/master-cron-job

```
bun run dev
```

2) Docker an K8S

~~Sorry, I spent two long evenings trying to containerize my apps and deliver them properly. While I still like Bun, it gave me a bit of fun and a punch at the end of the journey.~~

~~I planned to use Docker with:~~

~~turbo prune MY_APP --docker~~

~~But it requires Node.js. Some workarounds are possible (since Bun supports Yarn-style lock files), but they don't cover all cases, including mine.~~

~~I even tried to build everything into one and use it as a base (a wild idea), but again, some dependencies required rebuilding with Node, Python, GCC, etc. After waiting for over an hour, I stopped.~~

~~Maybe in the next couple of evenings, I'll try to fully migrate the apps to Node and deploy them in Docker/K8s. I'll send some updates then.~~

Playing around tools, versions, base images….

In distributed-scheduler@schedule-apps

docker compose up -d

UI should run on http://localhost:3000
By default master-cron-job will be executed every 3 minutes