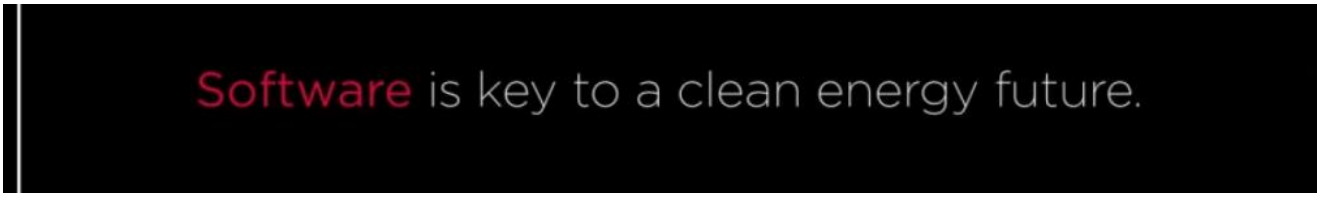
Colin Breck and Percy Link explore the evolution of Tesla's Virtual Power Plant (VPP) architecture. A VPP is a network of distributed energy-resources (often solar, wind, batteries) that are aggregated to provide smarter and more flexible power generation, distribution, and availability. Tesla's VPP consists of vertically integrated hardware and software, including both cloud and edge computing.

We now have alternative sources of energy different from the traditional, large power sources of the past

Software is the key to enabling all these diverse power sources and systems to act in concert

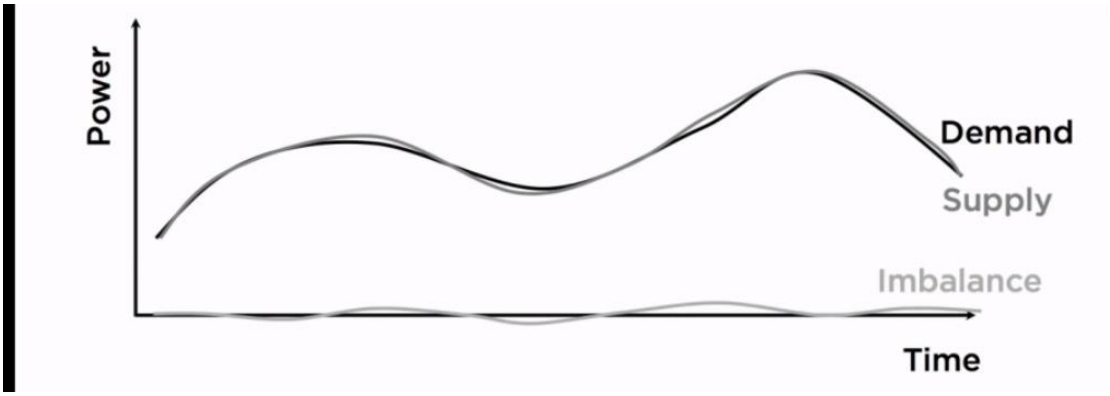
We can bring together thousands of all batteries in people's homes to create virtual power plants to provide value to energy grid and people's homes. This is one of the challenging problems in distributed computing today concerning distributed renewable energy.



We will see the evolution of the Tesla Virtual Power Plant and share architectures, patterns and practices for distributed computing and IoT that helped us tackle these complex challenges.

We built Tesla's optimization and market participation platform and the Cloud IoT platforms for Tesla's energy products

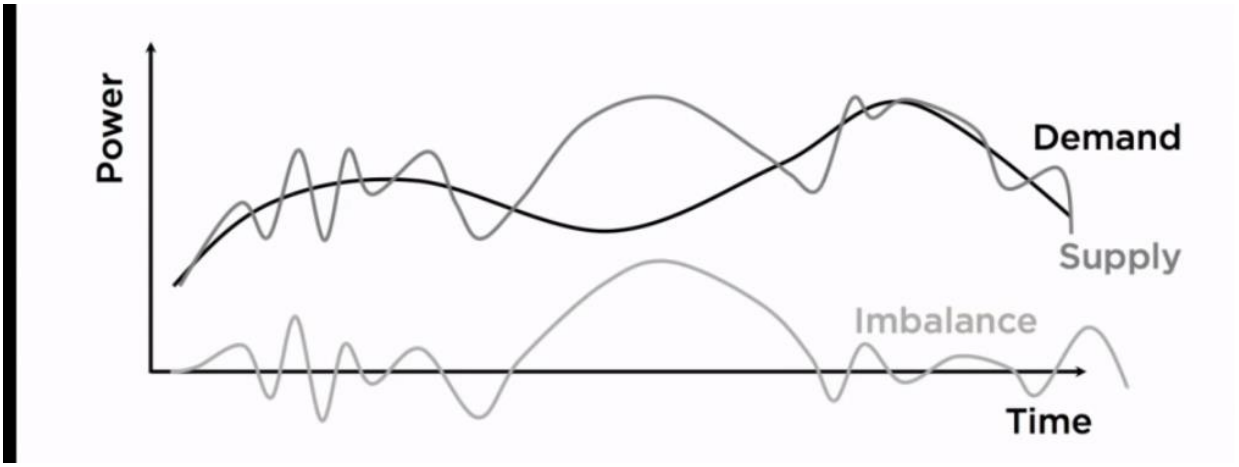
Let us see how the grid works and the role that batteries play in it so that we appreciate the software solutions



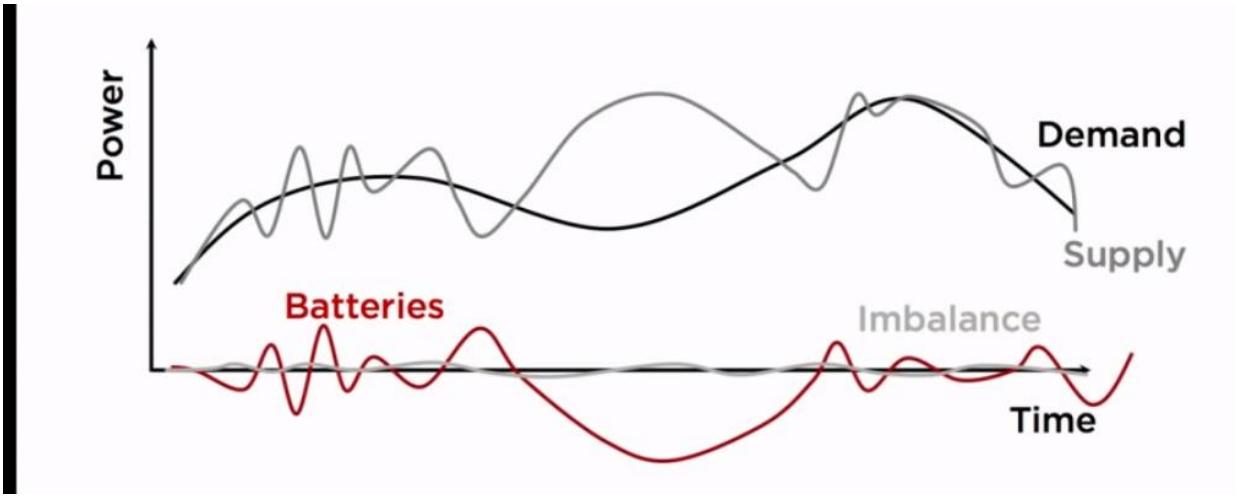
The tricky thing about the power grid is that the power demand and supply have to match in real time unless frequency and voltage can deviate and problems occur within the grid like damaging devices and causing blackouts. The grid itself has no ability to store power, so the incoming power supply and the outgoing power consumption need to be controlled in a way that maintains the balance.

With old style centralized fossil fuel power generation systems, supply can be turned up and down according to demand on the relatively small number of plants in operation. This made it relatively straight forward to maintain the balance.

As more renewable power generation comes into the grid, a few things happen like (1) reduced control, (2) uncertainty and rapid changes in generation leading to real-time forecasting changes and problems, (3) distribution is unpredictable since there are now many smaller generator units in homes adding energy to the grid independently.



The supply might look like above is a grid with large solar and wind energy generating units with variability in supply and times with high supply and low demand misalignments. This can result in power surplus and deficits that are larger than needed

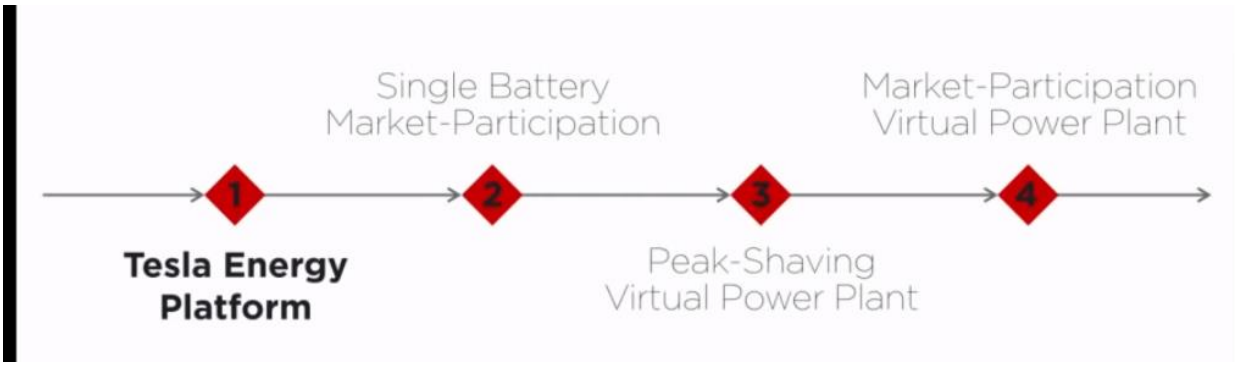


Batteries can help smoothen these by charging during the surpluses and discharged during deficits to quickly offset the rapid imbalances.

A solution could be to install giant batteries with large capacities like medium size coal plants

We can also take advantage of smaller batteries that are already being installed in millions of homes that are providing local value for things like backup power or helping owners consume more of their solar power generation.

We can aggregate these homes and small businesses with batteries and solar into Virtual Power Plants (VPPs) to provide cost savings and efficiencies



We start with the design and development of the Tesla energy platform, then how we learnt to participate in the energy markets and how we built the software to do this algorithmically using a single large virtual battery, then we discuss our first VPP where we learn to aggregate and directly control thousands of batteries in near real-time in people's homes to act as a single large battery, we then see how we combine all these platforms and experiences into how to aggregate, optimize and control thousands of batteries for energy market participation.

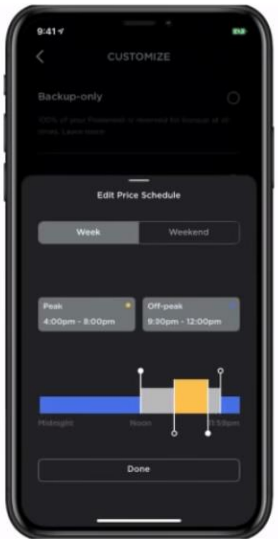
This Tesla Energy Platform **(TEP)** was built for both residential and industrial customers. For residential customers, the platform supports things like the Power Home Battery that can provide backup power for a house for hours or days in the event of a power outage.

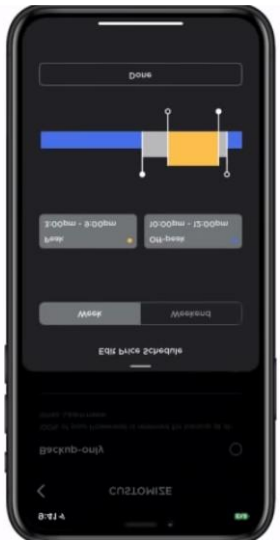
The second product supported by the TEP is Solar roof that also provides power for a house

Third are Retrofit solar panels for providing power to houses alone or paired with the battery pack to maximize solar energy produced.

We use software to deliver an integrated product experience across solar generation, energy storage, backup power, transportation and vehicle charging.

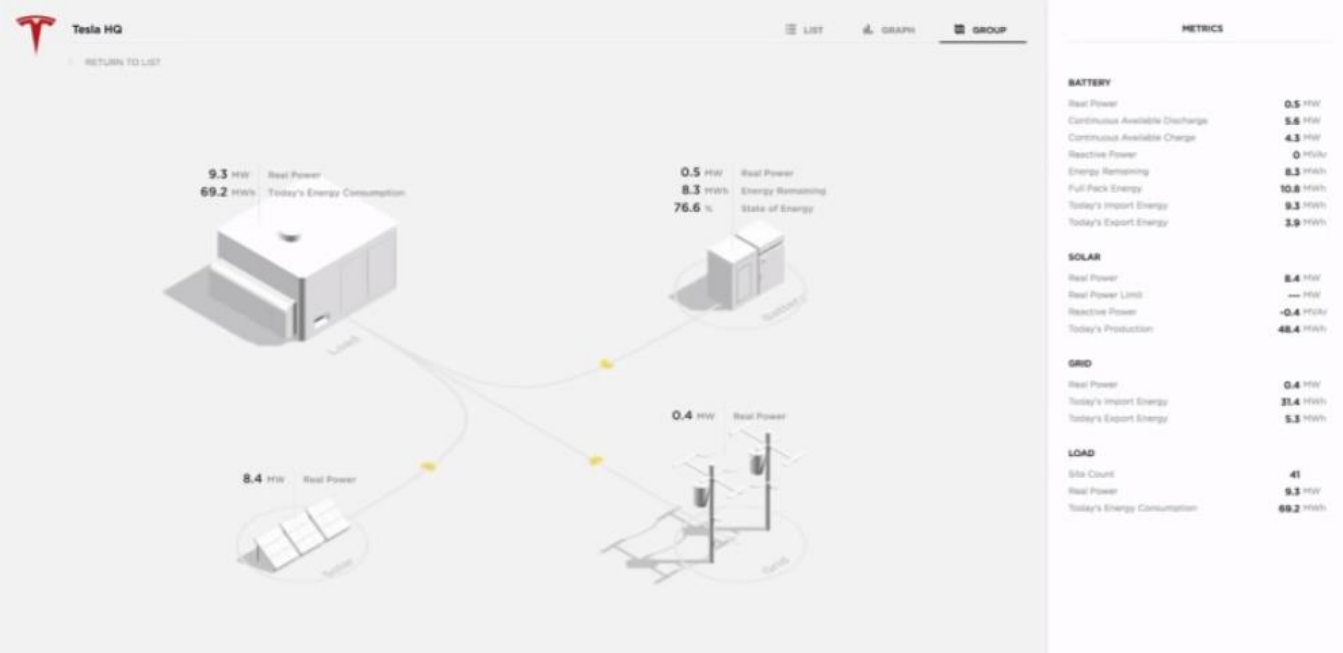
We also use software to create unique products like StormWatch, where you charge your Tesla PowerWall to full charge when notified of an approaching storm for emergency power cutoff scenarios.





Part of the customer experience is being able to view real-time status of the system on a mobile app. Customers can control some the system behavior such as specifying the charging times for low cost.

For industrial customers, TEP supports PowerPack and MegaPack for large scale energy storage, as well as industrial scale solar as below

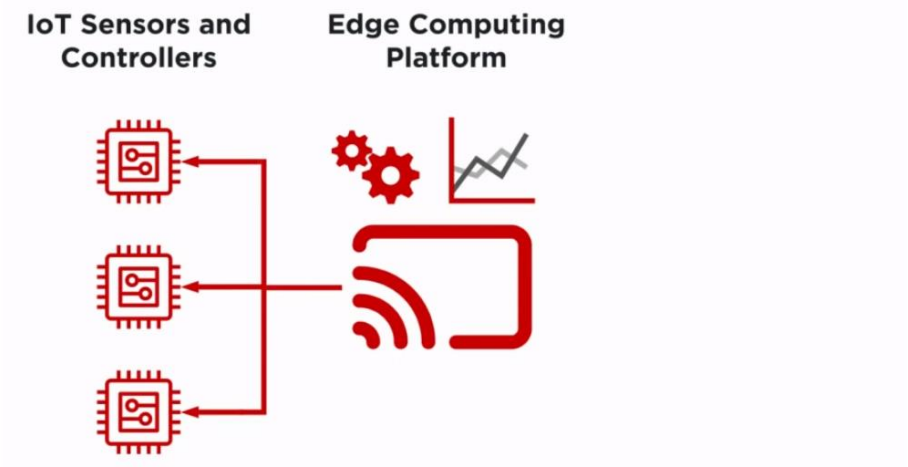


Software platforms like Power Hub allows customers to monitor the performance of their systems in real time.

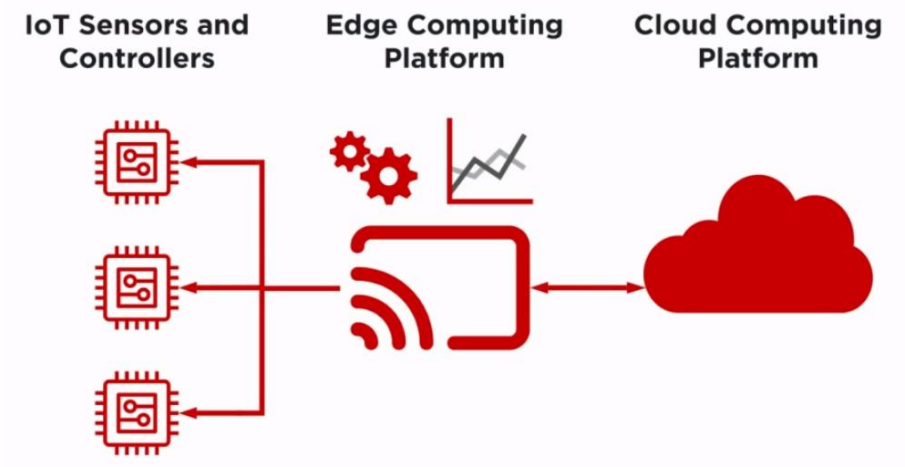
Or to inspect historical performance over days, weeks, or years

These products for energy generation, storage, transportation and charging all have an edge computing platform

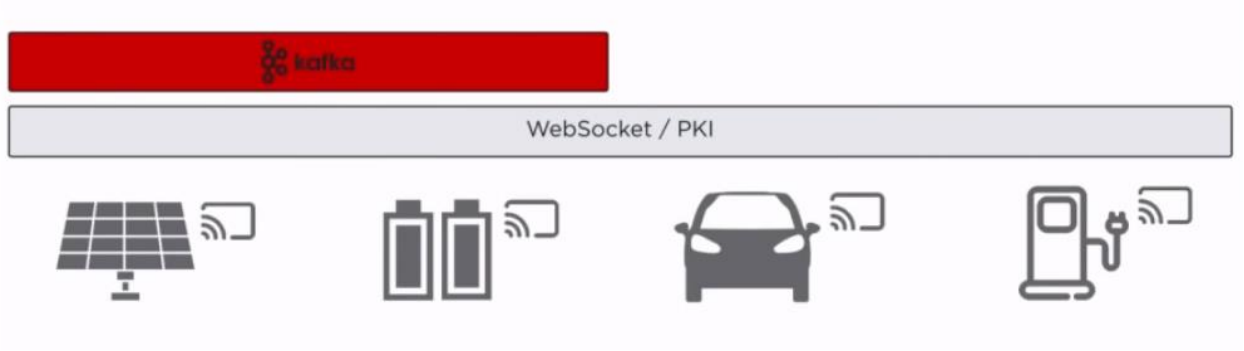
The edge computing platform **(ECP)** for energy is used to interface with a diverse set of sensors and controllers like inverters, bus controllers and power stages.



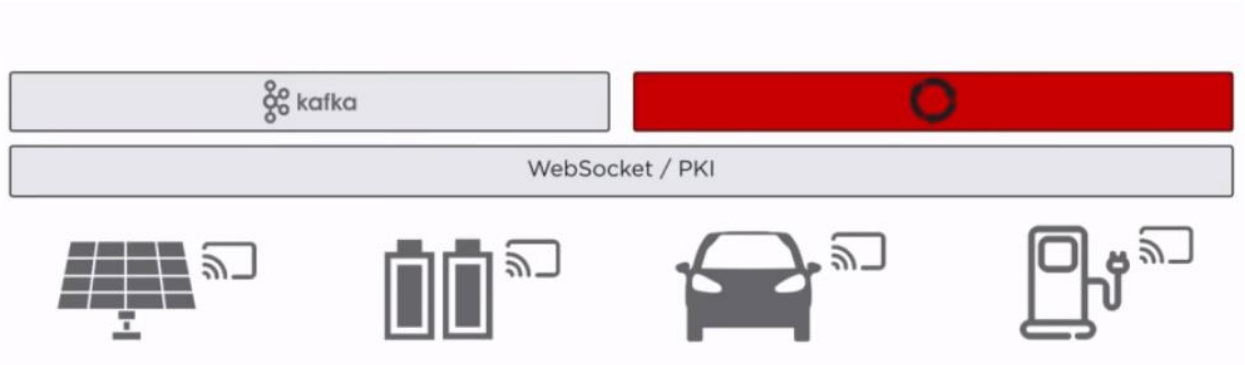
The ECP also runs a full Linux OS and provides local data storage, computation and control.



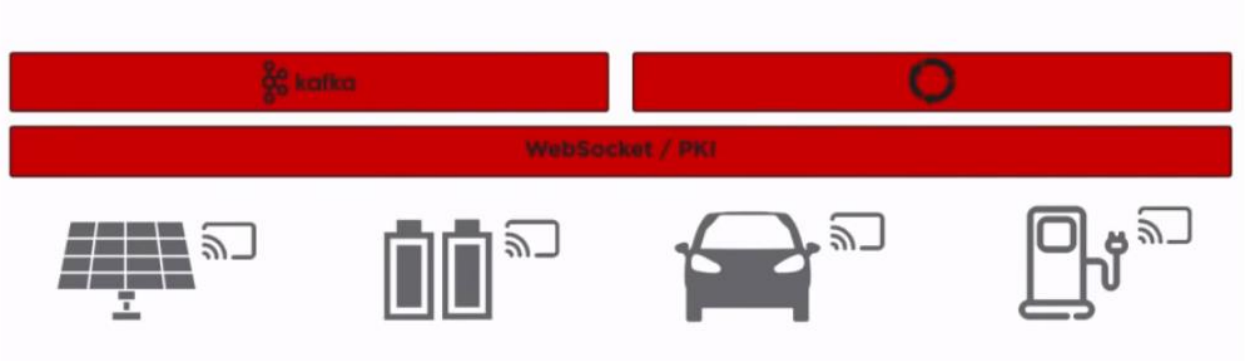
The **ECP** also **maintains a bi-directional streaming communication with the cloud over Websocket** so that it can regularly send measurements to the cloud as frequently as once a second, it can also be commanded on-demand from the cloud.



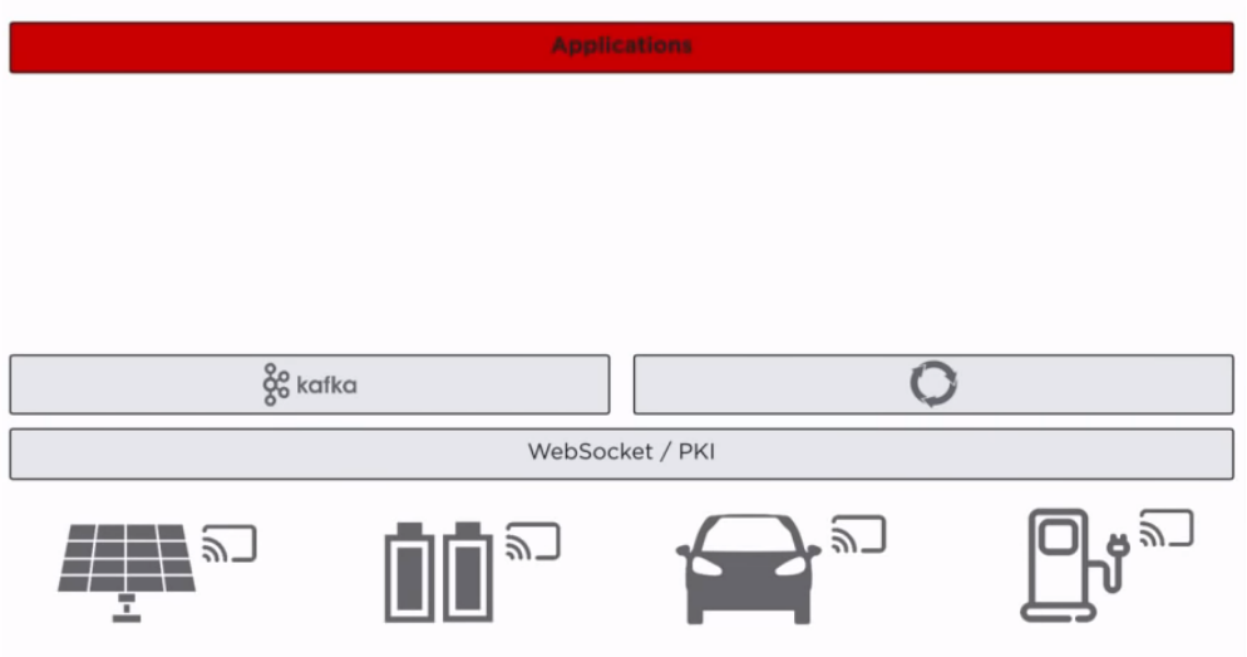
The foundation of the ECP is the linearly scalable **websocket** frontend that handles connectivity and security. It has a Kafka cluster behind it for ingesting large volumes of data from millions of IoT devices and provides message durability, decouples data publishers from data consumers, and sharing the telemetry across many downstream services.



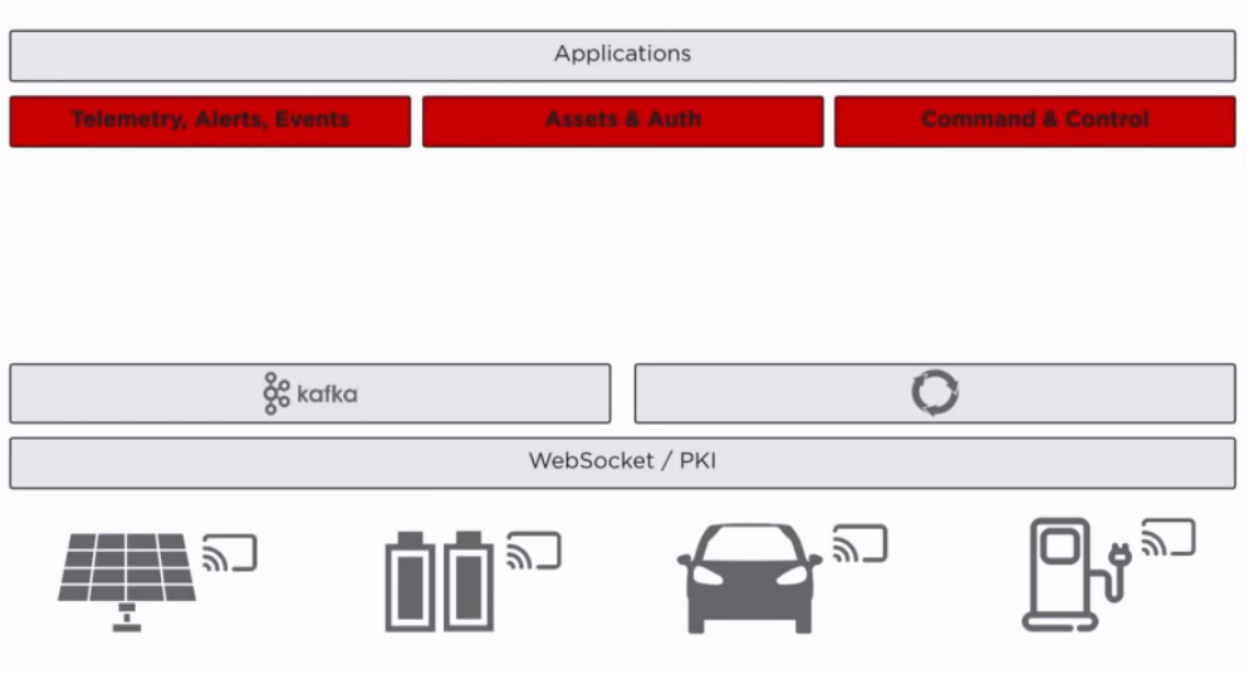
The ECP also has a platform for Publish/Subscribe messages to enable bi-directional command and control of IoT devices



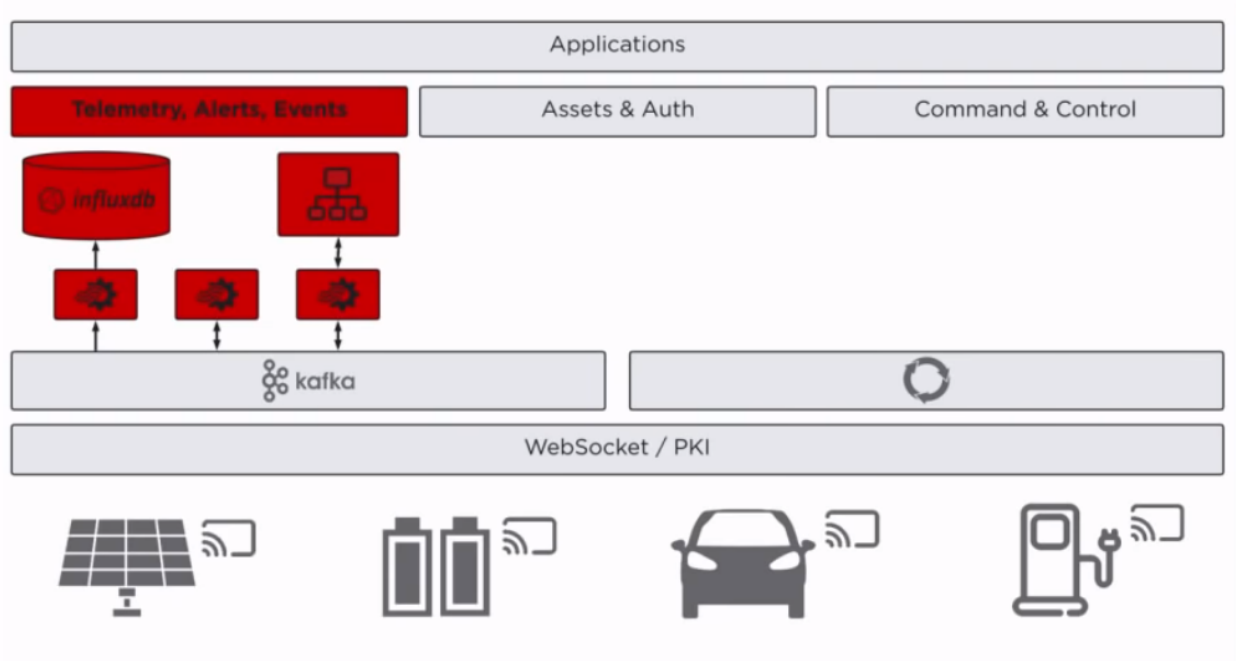
These 3 services together are offered as a shared infrastructure throughout Tesla for us to build high-order services on.



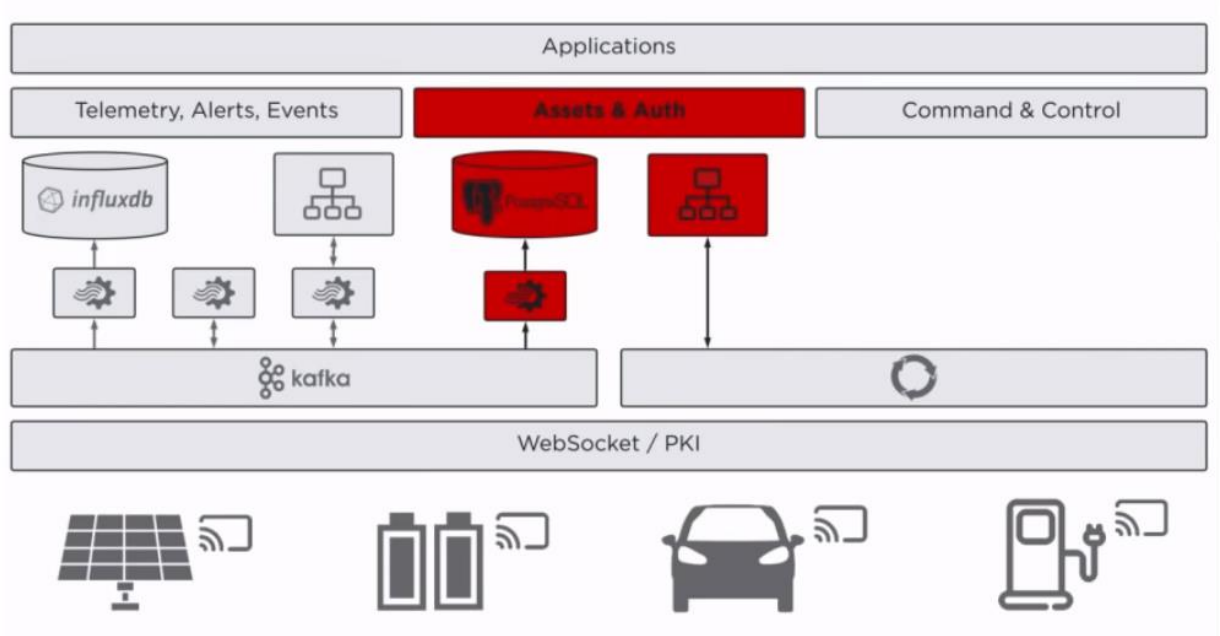
We then have several customer-facing applications supporting the functions mentioned earlier



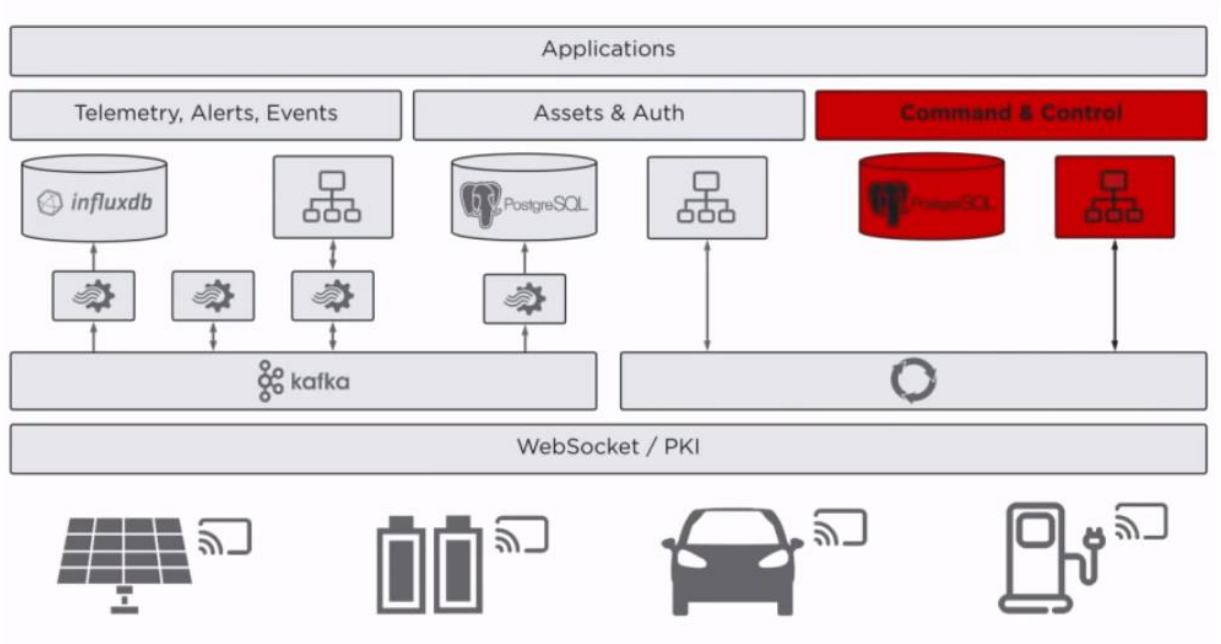
The APIs for energy products are organized broadly into 3 domains shown above. The APIs for querying Telemetry, Alerts, Events for devices or streaming these as they happen. Assets & Auth APIs are for describing energy assets and the relationships among the assets. Lastly, we have the APIs for commanding and controlling the devices like batteries.



The backing services for these APIs are composed of approximately 150 polyglot microservices in several domains. We use **influxDB**, **a time-series database for storing streaming IoT telemetry data from thousands of batteries as historical data**, this uses topics as they come in.



Product and customer data come from many different systems and are stored in our PostgresSQL databases for asset management. We do a lot of digital twin modelling to represent the current state and relationships of real assets.



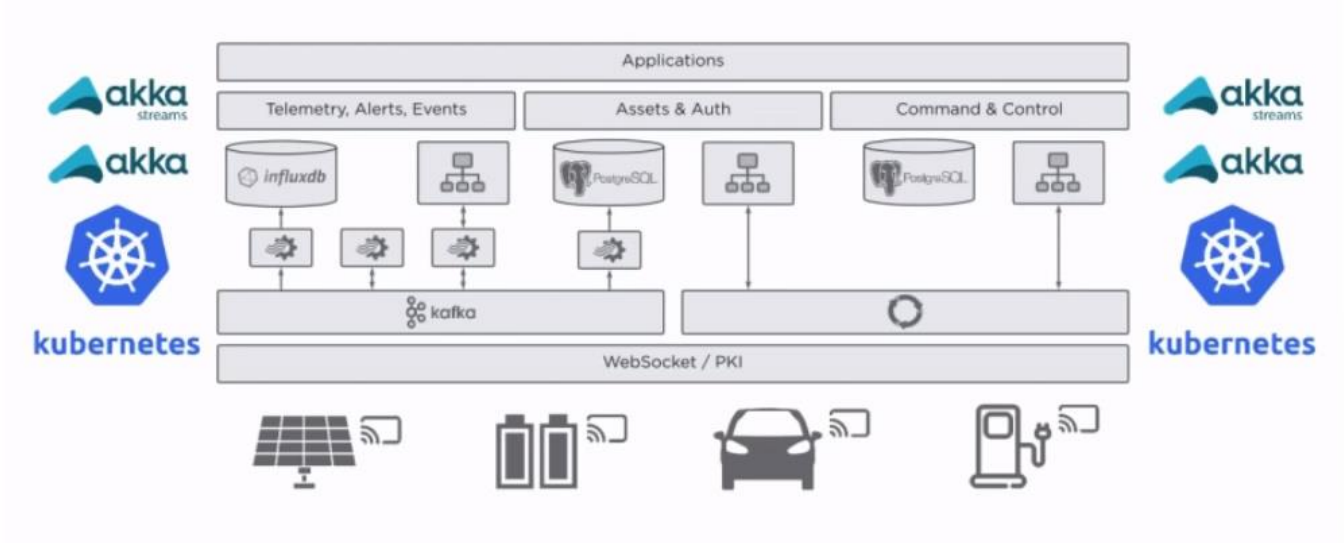
Command & Control services are used for finding and controlling IoT devices like telling a battery to discharge at a specific power setpoint for a specific duration. We need a streaming stateful, real-time representation of IoT devices at scale including modelling inherent issues with trying to control IoT devices across the internet.

Akka has been essential for us when building these microservices, it is a **toolkit for distributed computing** and it also supports **actor model programming** that is great for modelling the state of individual entities like a battery, while also providing a model for concurrency and distribution based on asynchronous, immutable message passing. Akka is a great model for IoT.

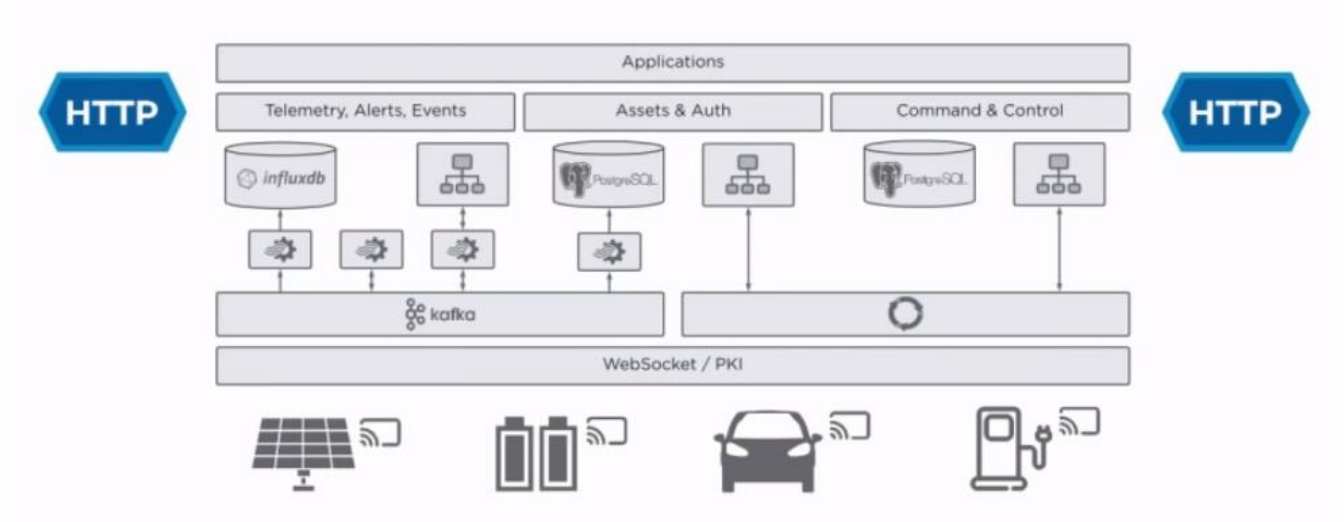
Akka Streams, the reactive streams component of the Akka toolkit is also used extensively. It provides sophisticated primitives for flow control, concurrency, and data management, all with back-pressure under the hood to ensure that the services have bounded resource constraints. Generally, all the developer writes are functions and then Akka streams handles the system dynamics that allows processes to bend and stretch as the load of the system and messaging volume changes.

The alpakka project has a large number of these reactive streams interfaces to services like Kafka, AWS S3, etc. this is what we use for interfacing with Kafka extensively, because the interface provided by Kafka streams is too simplistic for our use-cases. Akka streams provides a more general-purpose streaming tool that we use instead.

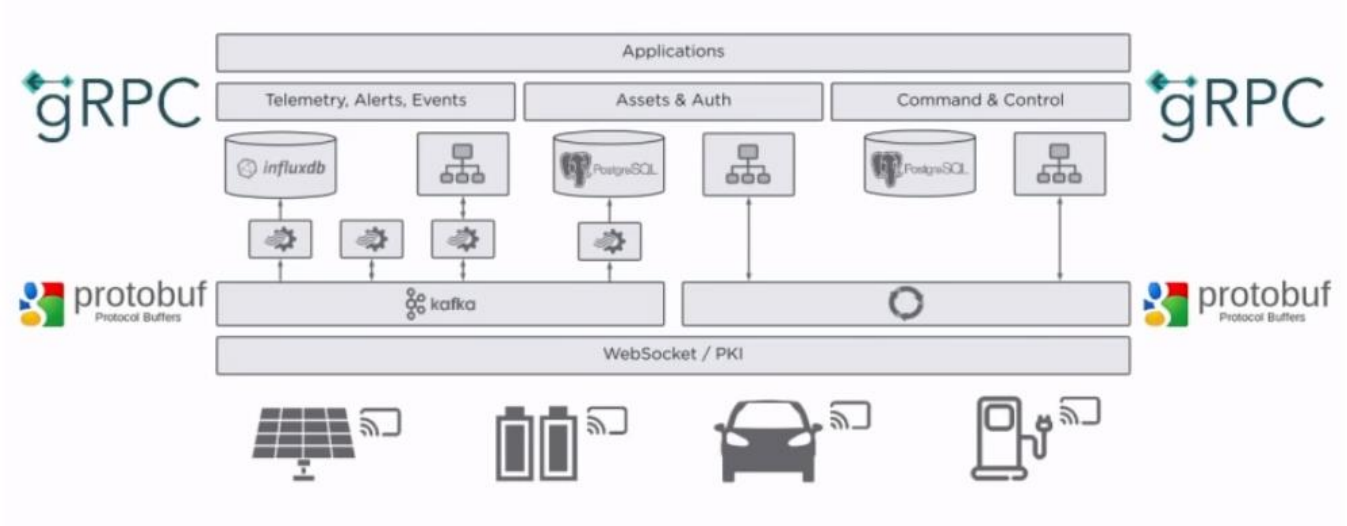
Scala is our primary programming language since it works well with Akka and functional programming for large systems



Most of our microservices run on **K8s**. We use **Akka** for fine grained failures like circuit breaking, retrying an individual request, modelling the state of an individual entity like a battery that is charging or discharging. We use **Akka streams** for handling the system dynamics and the message-based, real-time streaming systems.

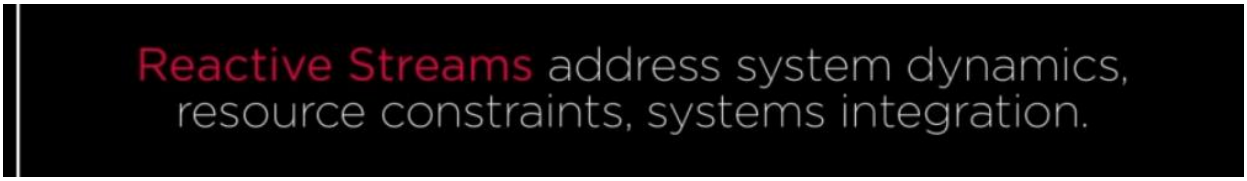


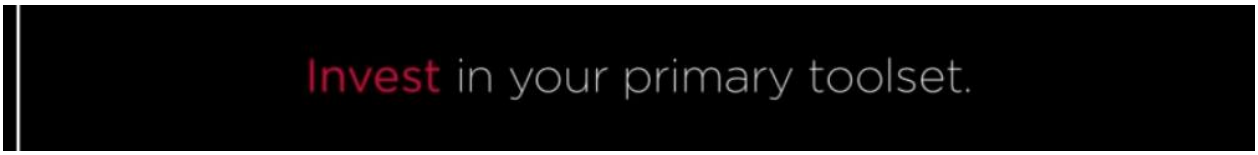
Our initial platform was built using traditional HTTP JSON APIs

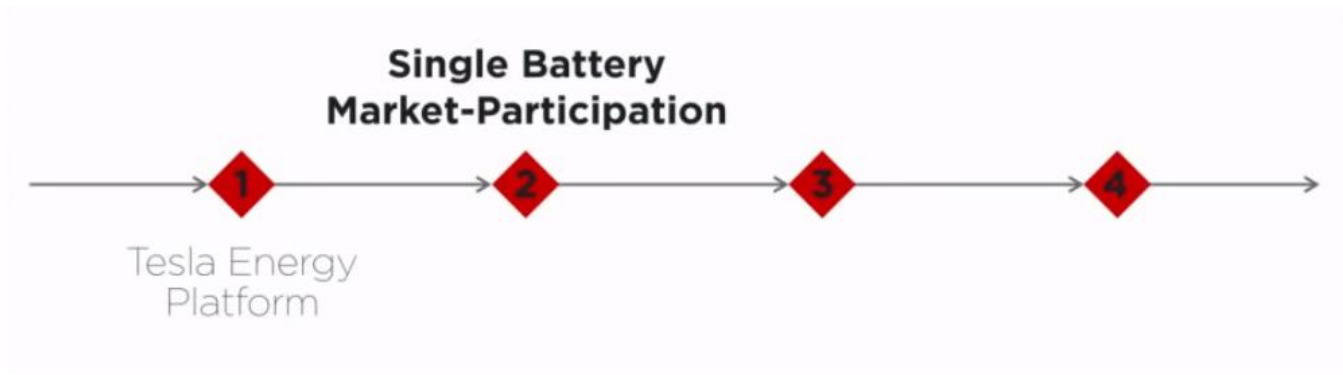


We have started changing new services to **gRPC** for **strict contracts**, **code generation of clients** is now possible and we no longer need to write the clients ourselves. We also use **ProtoBuf for our streaming data** while maintaining a single repository for sharing the contracts and collaborate across projects.

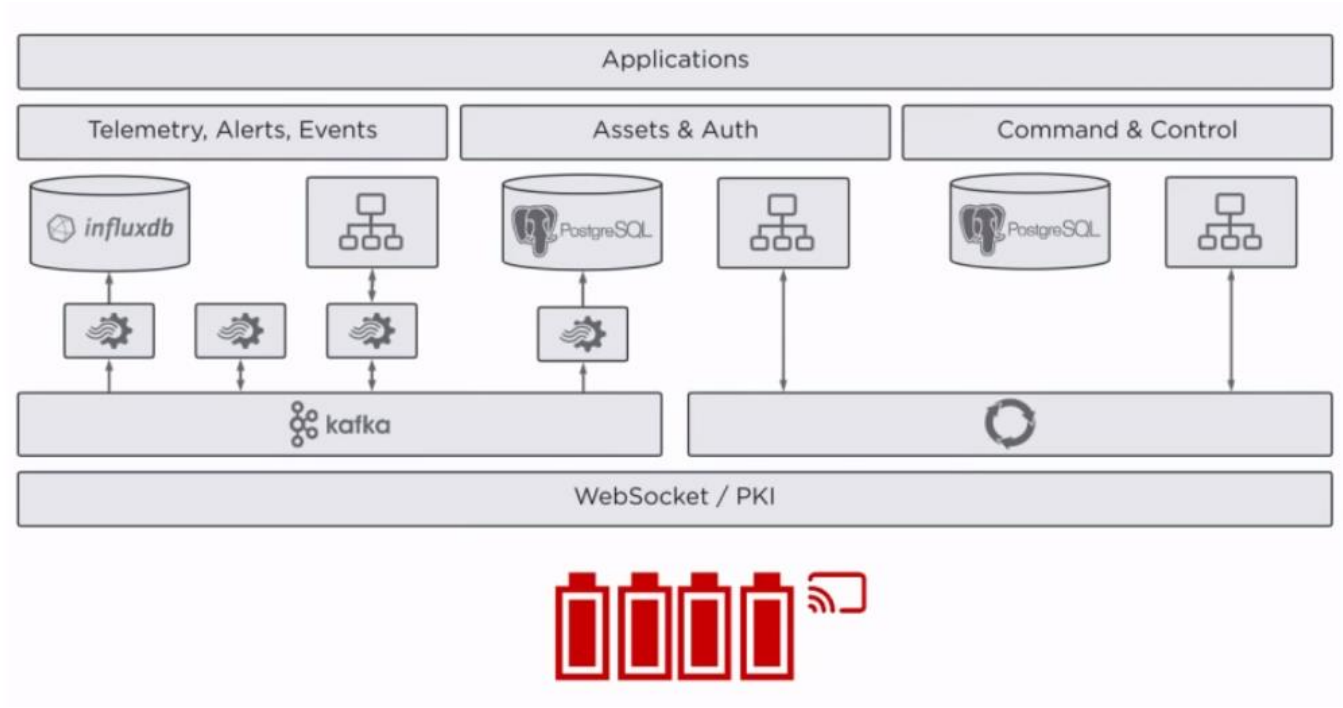
Rich typing in Scala, Strict schema with Protobuf, and Strict Asset Models for systems integration.







We have built our first power plant type application on top of the TEP



In this phase, we are learning how to productize real-time forecast and optimization of batteries. We started with a single, very large battery.

This is the largest battery in the world at Hornsdale with 100MW at 129MWhrs, it has the same power output as a giant gas turbine

FINANCIAL REVIEW

South Australia's big battery slashes $40m from grid control costs in first year

Angela Macdonald-Smith
Senior Resources Writer

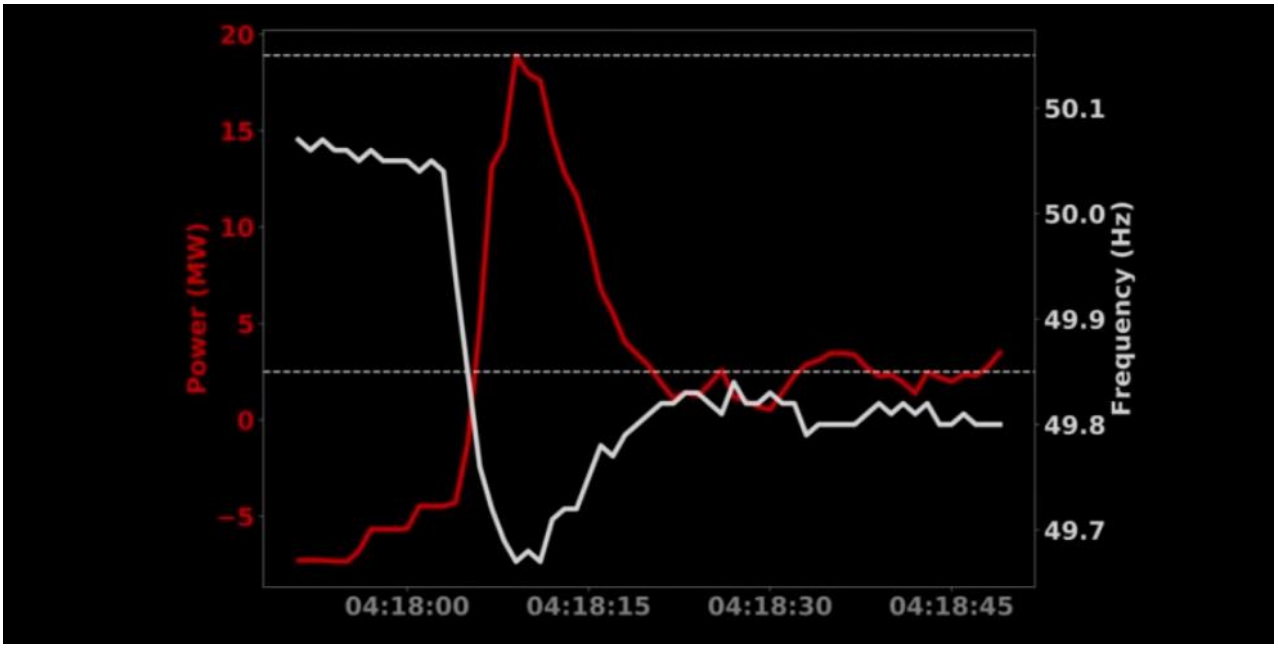
Updated Dec 5, 2018 — 8.10pm,
first published at 8.24am

Save Share

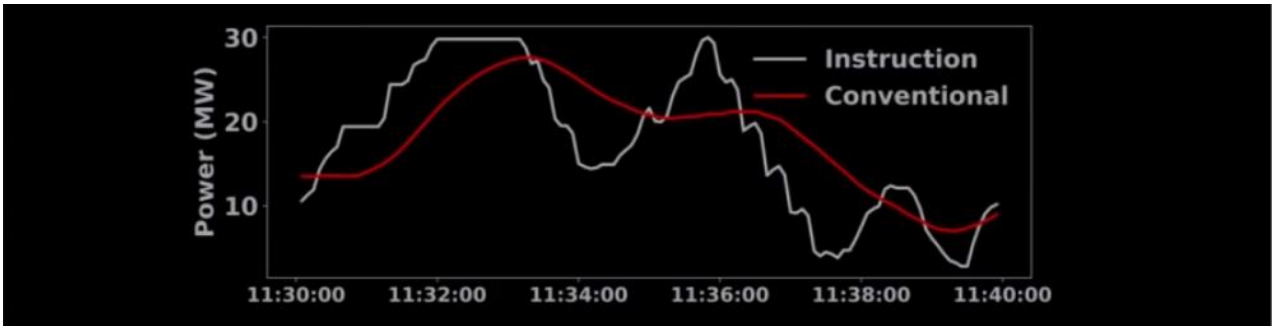
South Australia's big battery has outperformed expectations in its first year of operation, saving almost $40 million in grid stabilisation costs, helping prevent blackouts, and generally restoring confidence in energy supply in the state, according to French project backer Neoen, which is working on expanding its battery project line-up in Australia.

The 100 megawatt Hornsdale battery, the idea for which sparked from a famed Twitter exchange between tech billionaires Mike Cannon-Brookes and Tesla's Elon Musk, has had a huge impact in reducing prices in the grid stabilisation market, known as the frequency control and ancillary services (FCAS) market, according to an independent report released on Tuesday by engineering and advisory firm Aurecon.

This giant battery is used to help keep the grid stable even as more renewable power sources are coming online and reduces the customer power costs

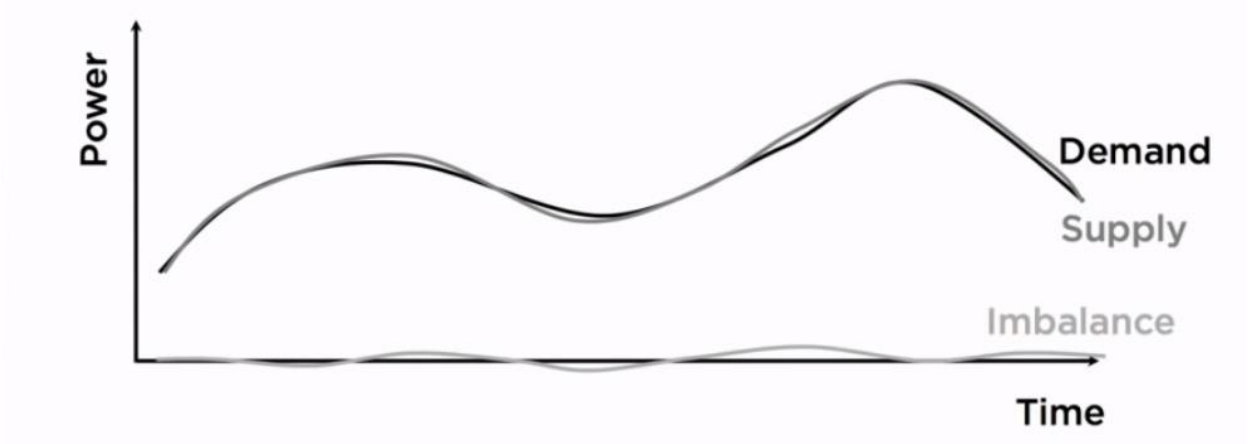


During a generator downtime event, the **Hornsdale battery** responds near instantaneously to fill the power gap and prevents frequency excursions and blackouts



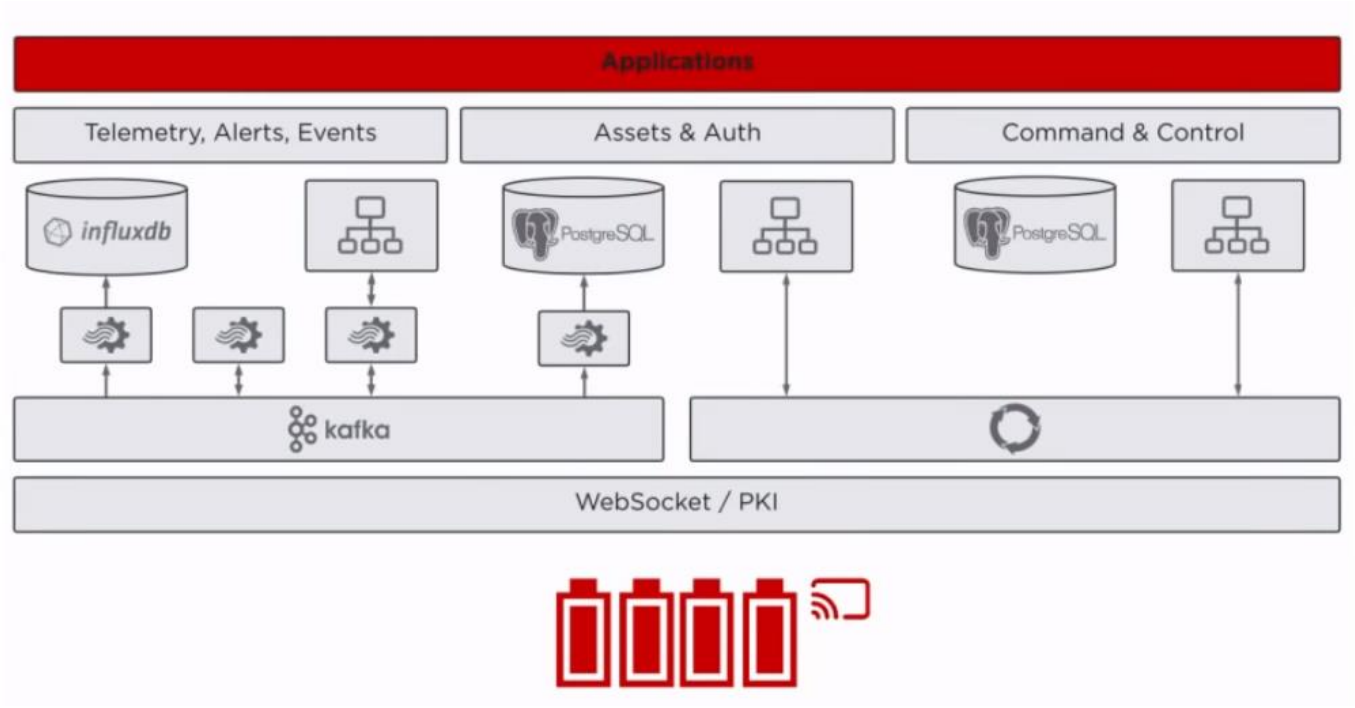
Even during normal generation times, the battery helps smoothen out power generation fluctuations

This battery provides these services to the grid through participation in the energy markets

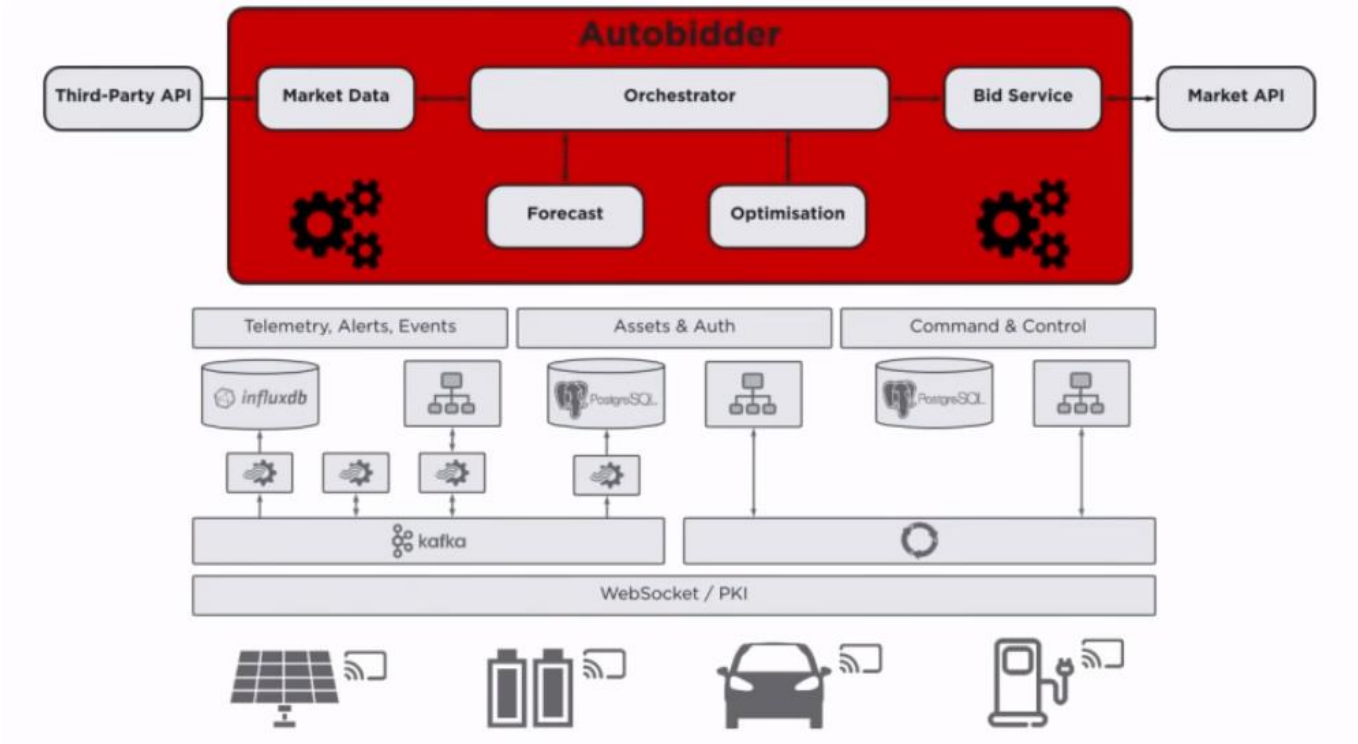


Markets are the way to make sure that the power demands and supply are balanced every time. Participants bid in what they are able to supply or consume at different price levels at different times. An operator activates the participants who can provide the necessary services at the lowest price.

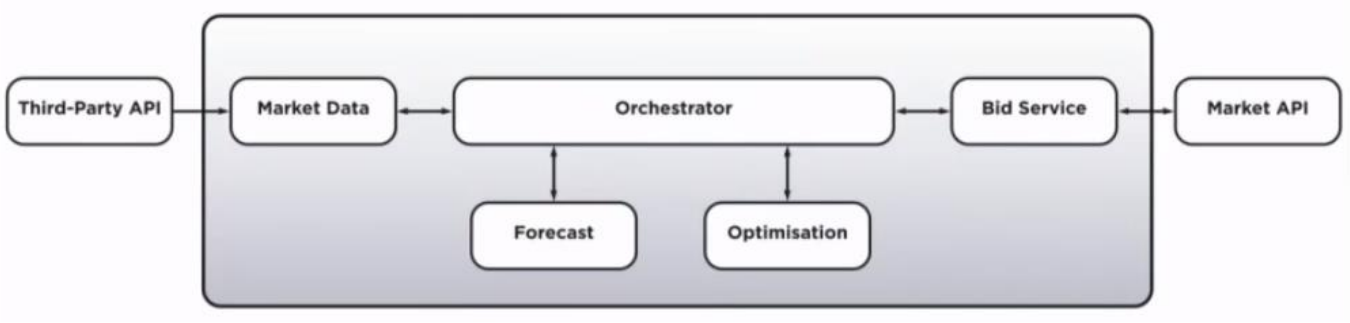
We built a software called **AutoBidder** to operate the **Hornsdale battery** and operate as a product. It runs workflows that fetches data, forecast prices, get renewable power generation data available to be supplied to the battery, decide an optimal bid for submission. These workflows run every 5 mins to match the cadence of the Australian power market. At a high level, the optimization problem is a tradeoff across different market segments for the different kinds of services that the battery can provide across time since the battery has a finite amount of stored energy.



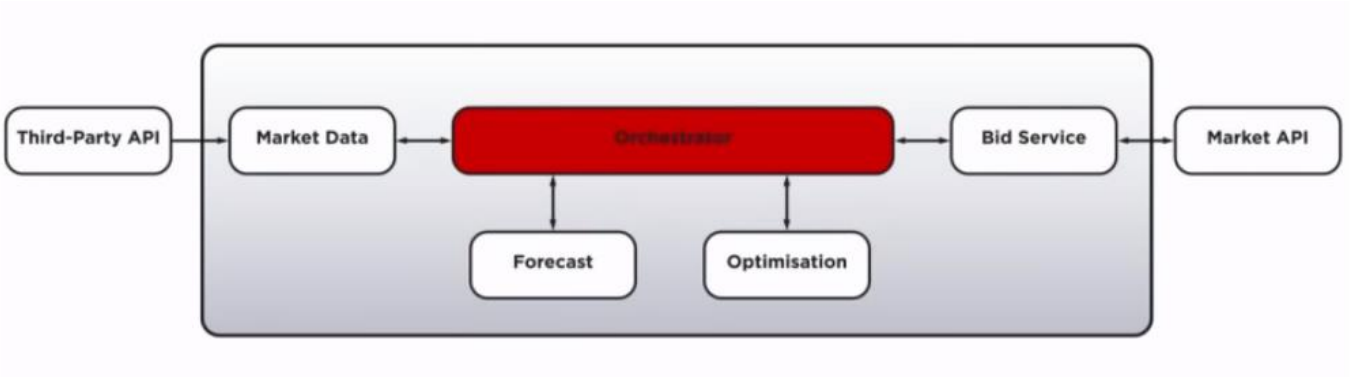
AutoBidder is built in the applications layer within the TEP



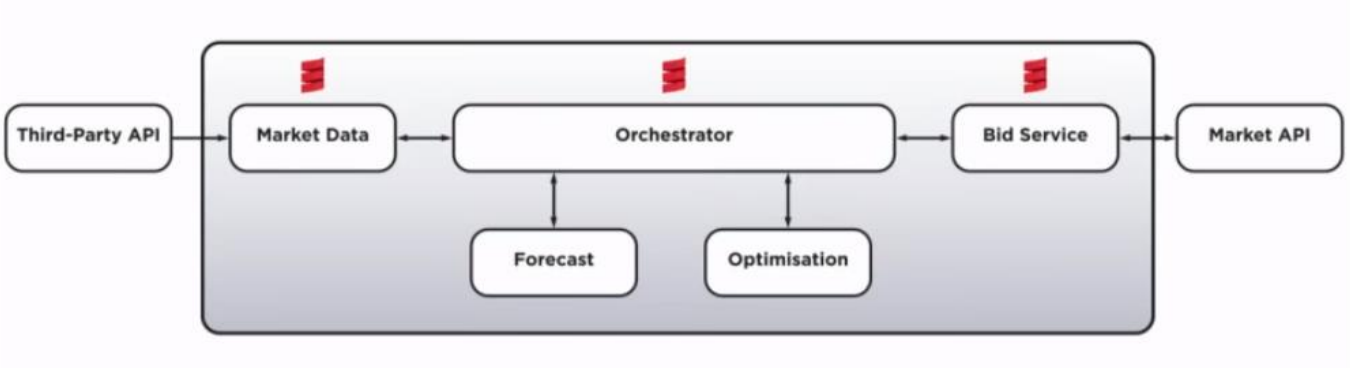
AutoBidder consists of several microservices and interact with both the platform and third-party APIs.



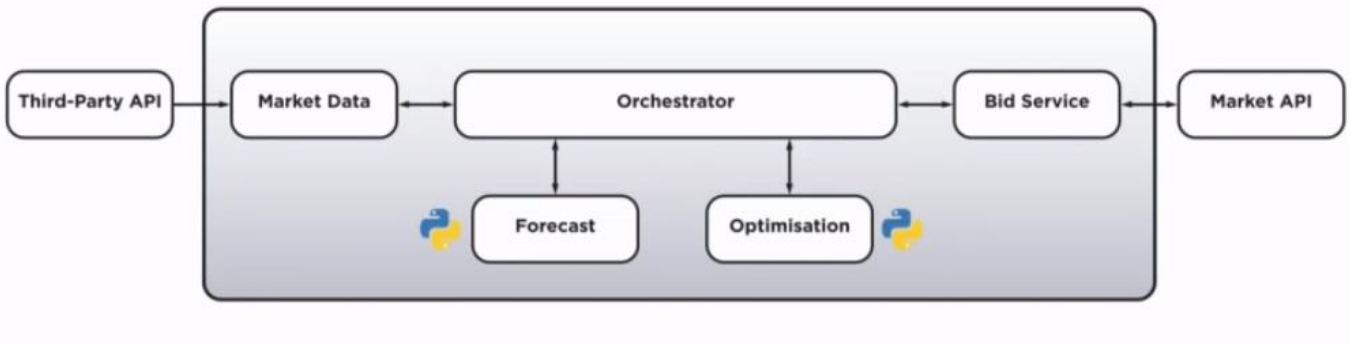
AutoBidder is simply a workflow orchestration platform as operational technology that leverages our primary toolset.



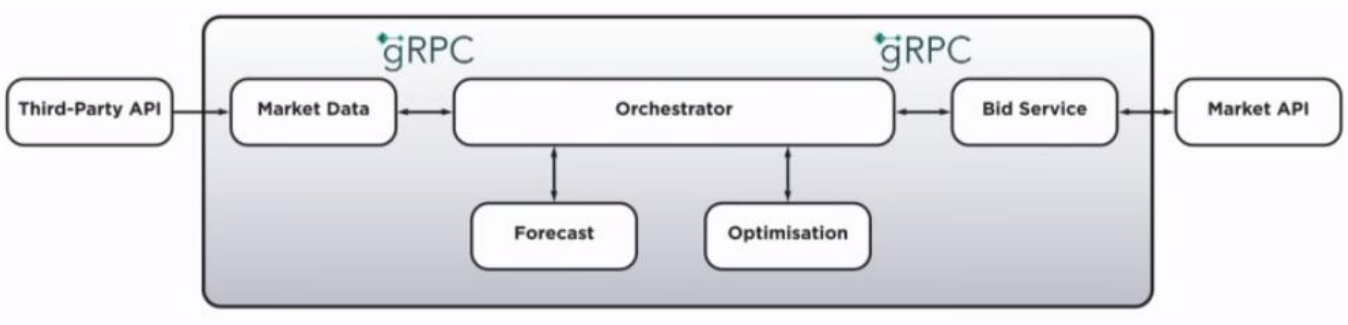
The **Orchestrator** service is the core that runs the several autobidding workflows. The **Market Data** service abstracts the ETL complex into data that has diverse timing arrivals relative to the market cycle, this service handles the timing and late arriving or missing data issues. The **Forecast** service and **Optimization** service simply execute algorithmic code. The **Bid Service** interacts with the market bid submission interfaces to the Market APIs.



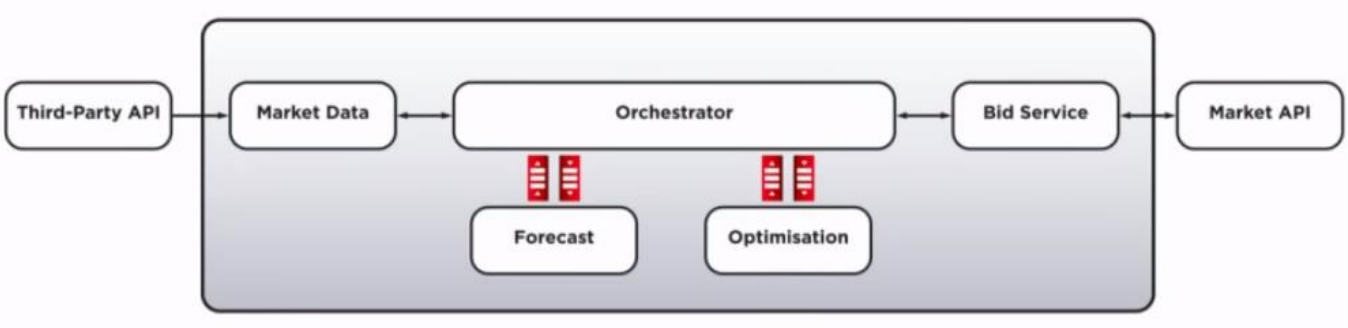
We use Scala because of the functional programming and type safety available in the language



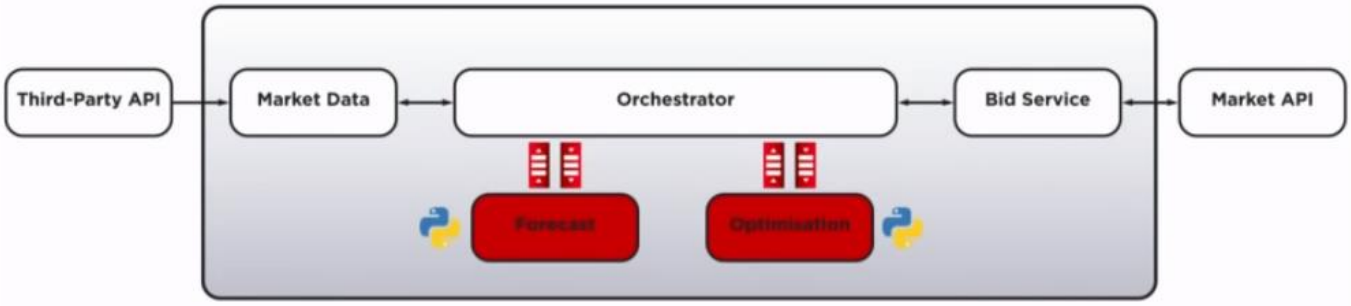
We use Python because of the key numerical algorithmic libraries available for doing computational work and rapid improvements



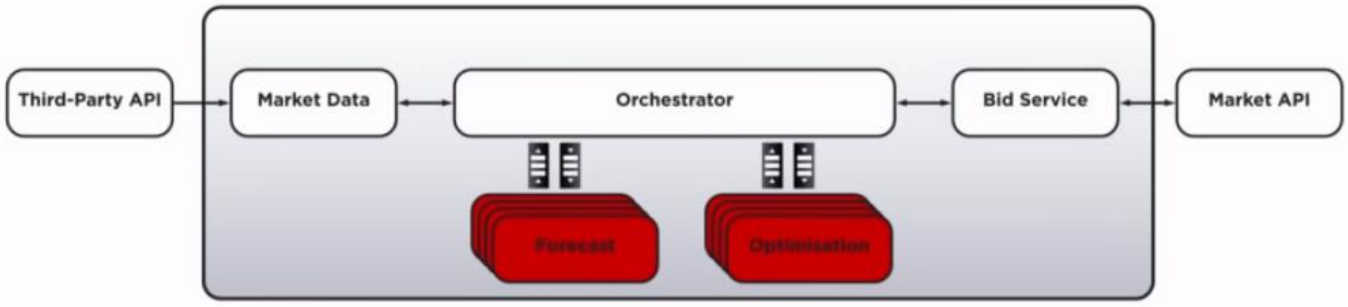
The communication between the market data and bidding services with the Orchestrator happens over gRPC for the benefits like strict contracts, code generation, and collaboration.



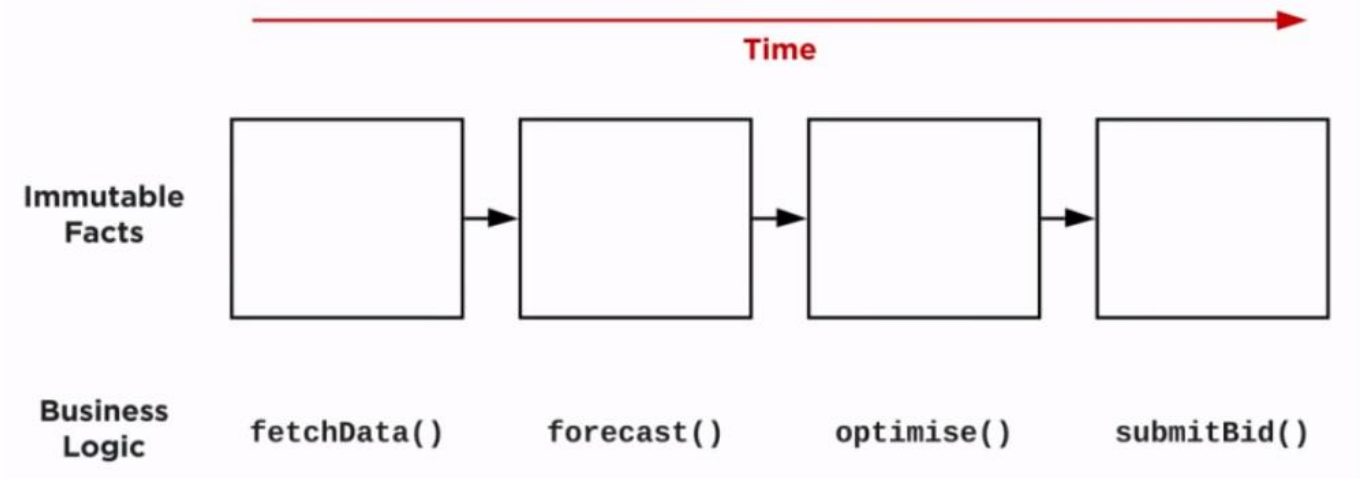
The communication between the Orchestrator and the Forecast and Optimization services uses Amazon SQS message queues for durable delivery, retries in cases of consumer failures, and supporting long running tasks without the support of a long running connection between the services. We use an immutable input-output messaging model, the messages have strict schemas that allows us to persist the immutable inputs and outputs and have them available for back testing.



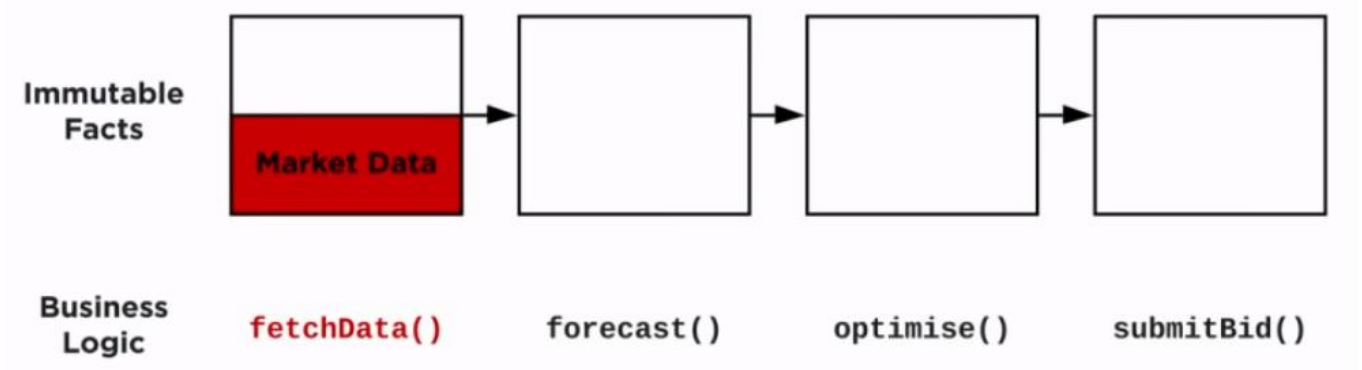
SQS also allows us to build worker pools that run the many algorithmic workflows (with concurrency semantics) needed by these services.



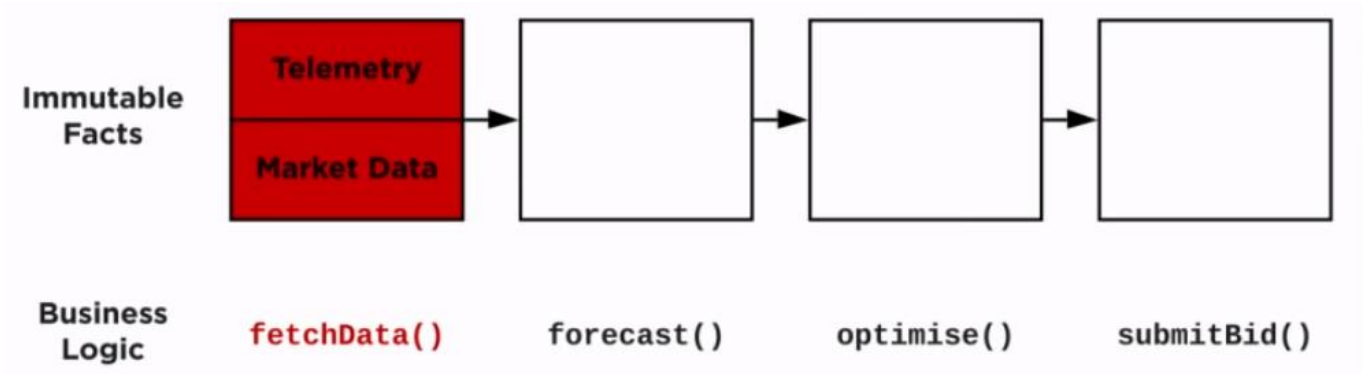
The **SQS message queues** allow us to implement **concurrency** across the workers instead of within a worker. These services are essentially functions that take inputs and produce outputs without other effects. This makes them testable, makes **algorithmic improvements** safer to introduce, and relives our algorithm engineers the burden of writing I/O code and use Scala concurrency for I/O.



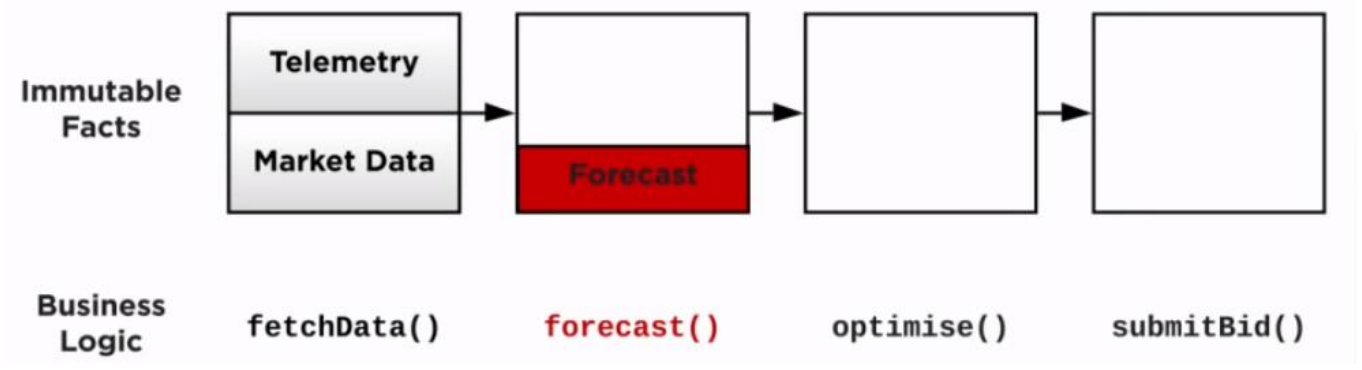
The workflows are stateful and the state is a set of immutable facts that are generated by business logic stages. The stages in the workflow happens sequentially in time.

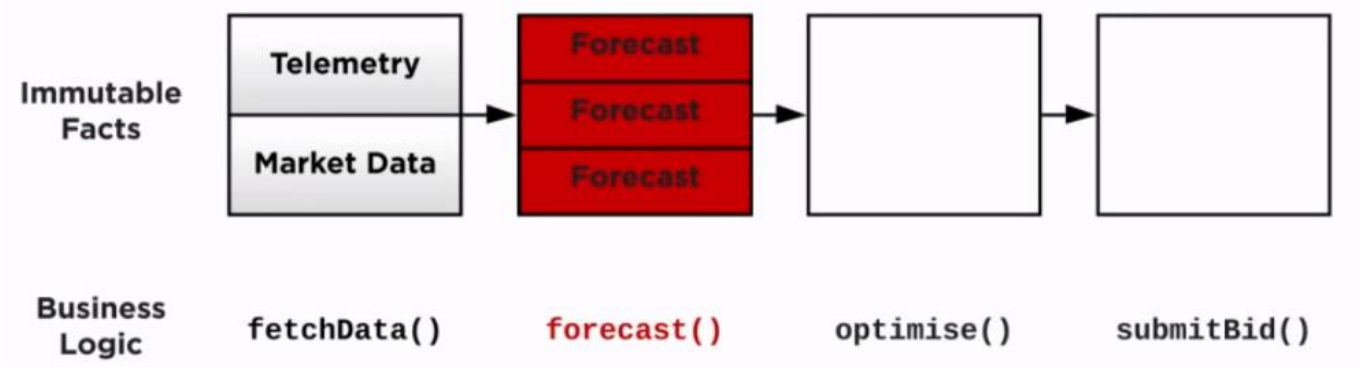


The workflow includes things like knowing what the current stage is

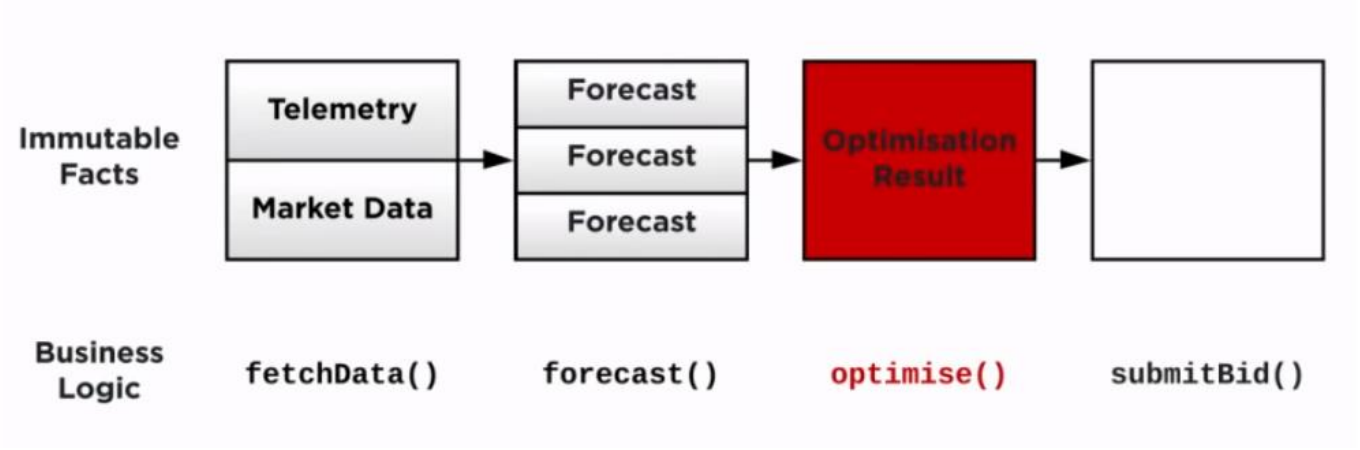


As well as accumulating the results of the task within the current stage and across stages

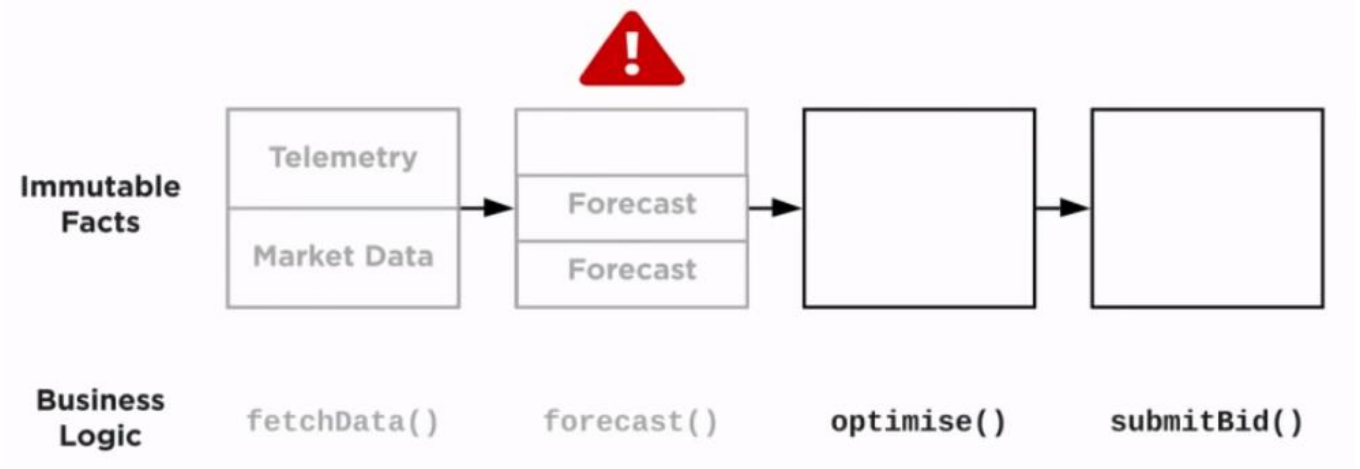




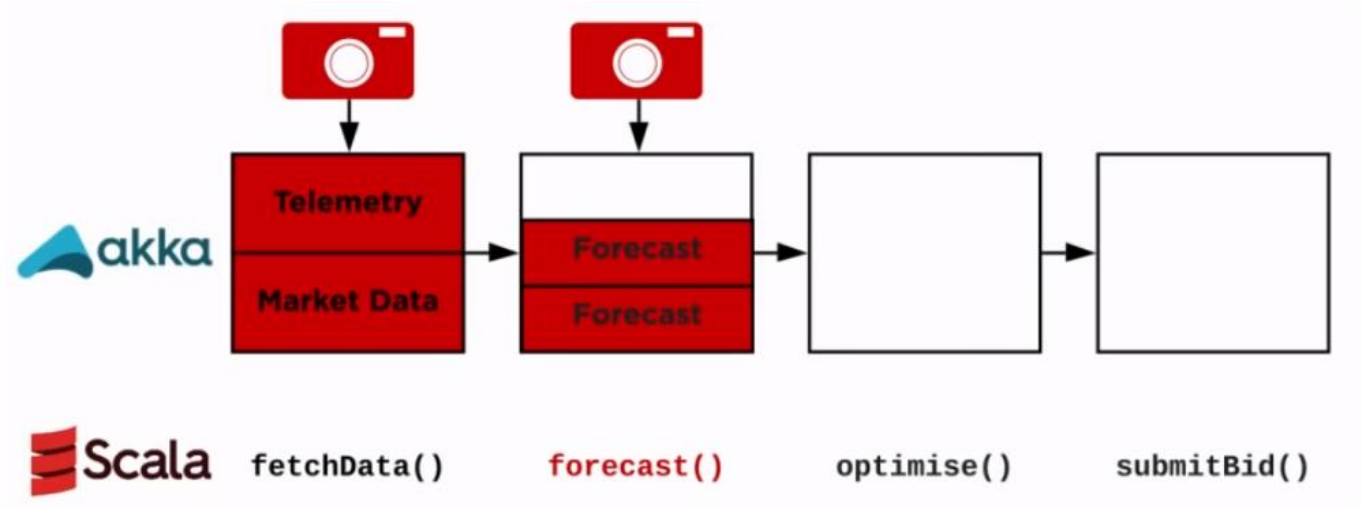
Some stages like the **forecast()** stage have multiple tasks that need to be accumulated before deciding to proceed



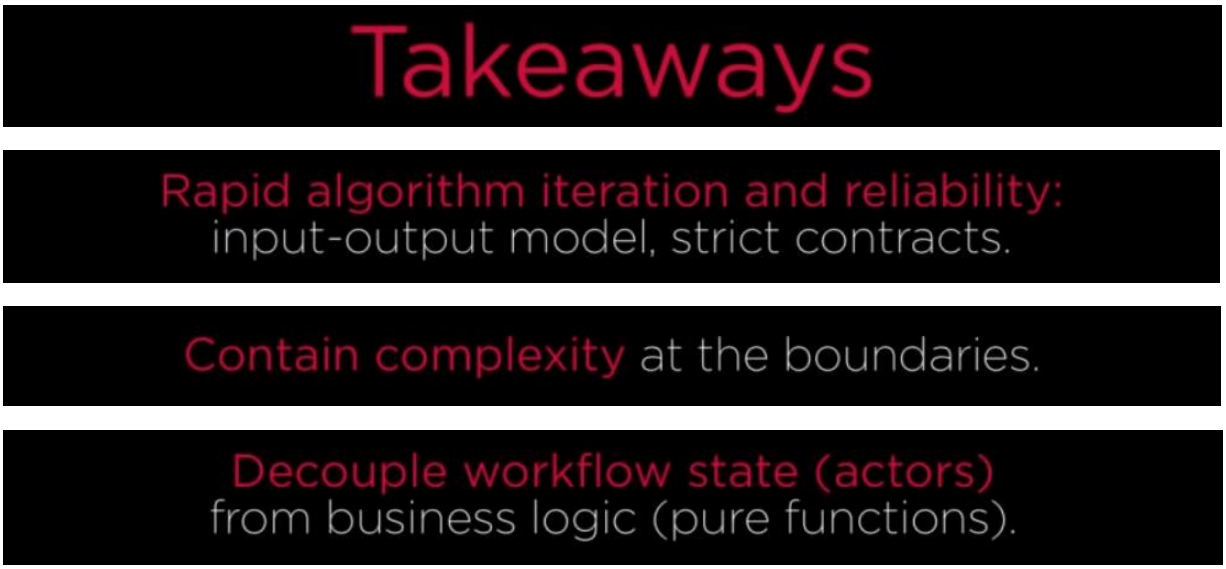
Some stage like the optimization result stage might need outputs of multiple previous stages and not only the predecessor stage

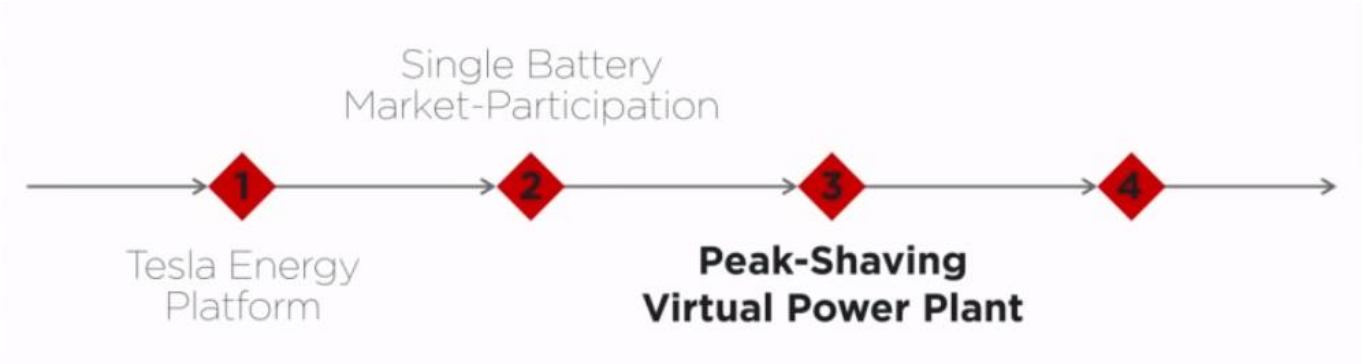


In case of a failure like when the **Orchestrator** restarts, we don't want to forget that a workflow was in progress. We do not want to completely restart the running work flows altogether

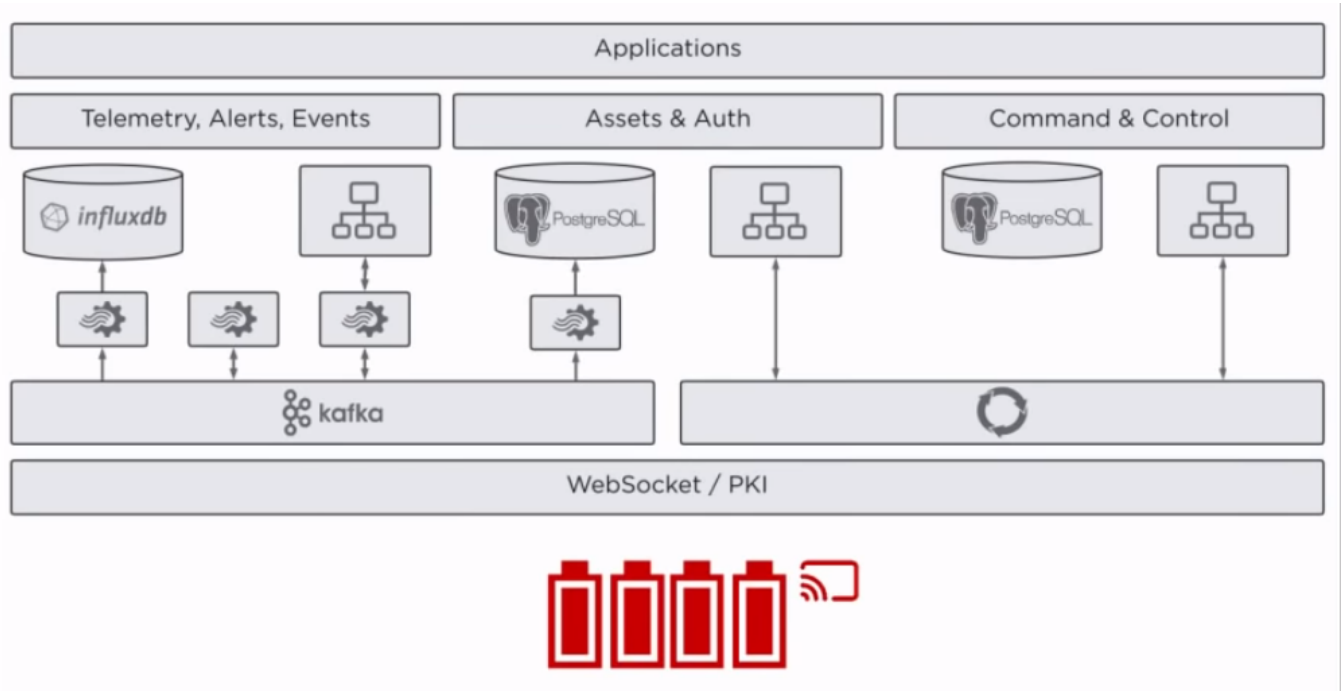


We instead take **periodic snapshots of the data** as checkpoints during the stages and can choose them as a starting point if needed during workflow and service failures. We keep the state in an Akka **Actor** that represents the state of the workflow state machine. **Akka persistence** gives us transparent state resumption through checkpointing and an event journal. We MUST keep the business logic of state executions as pure functions separate from the actors, this makes the testing and composition of those business functions easier using the new Akka **Type API**.

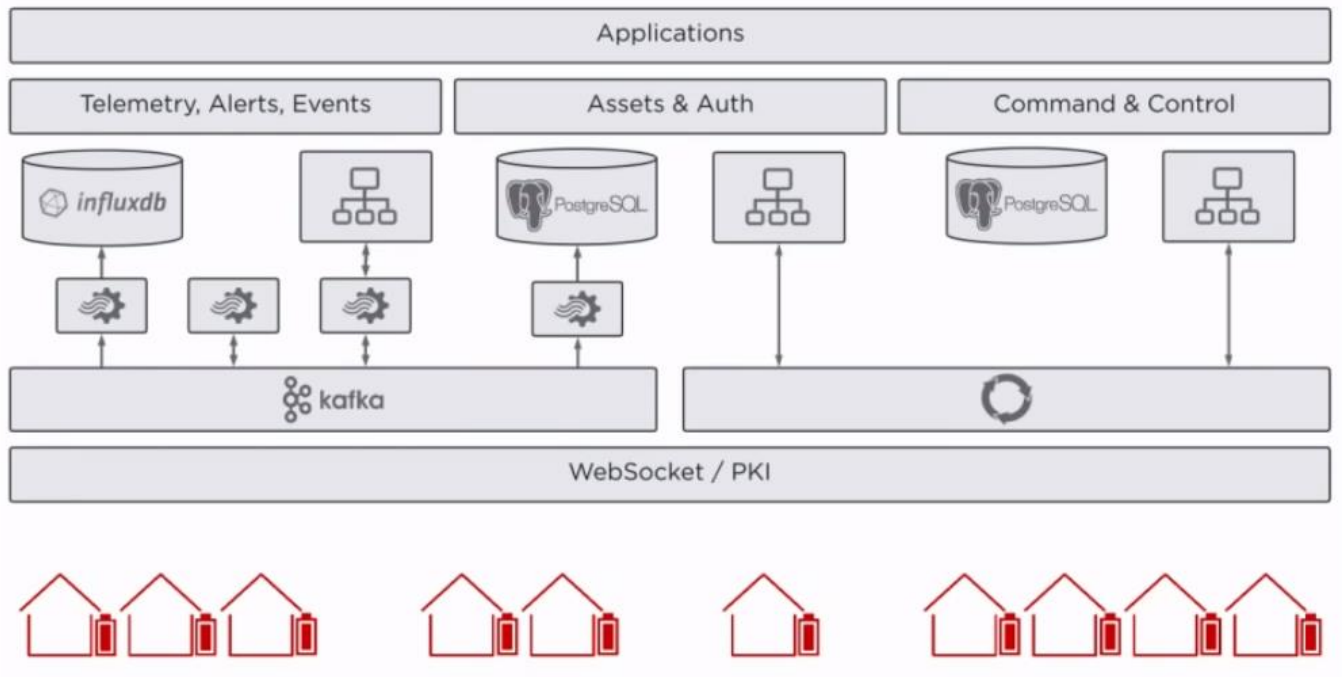




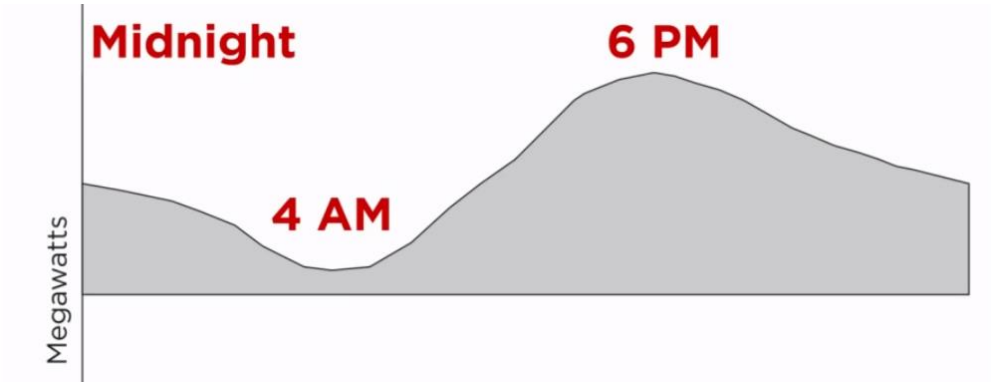
We will now describe our first VPP.



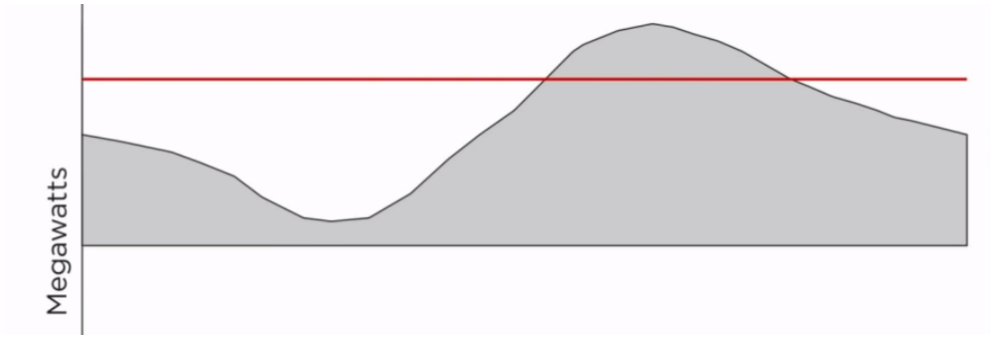
We have seen how we ***participate in the market algorithmically*** using one large battery, we can now use what we are learning to measure model and control a fleet of thousands of PowerWalls installed in homes to do peak shaving for an electrical utility.



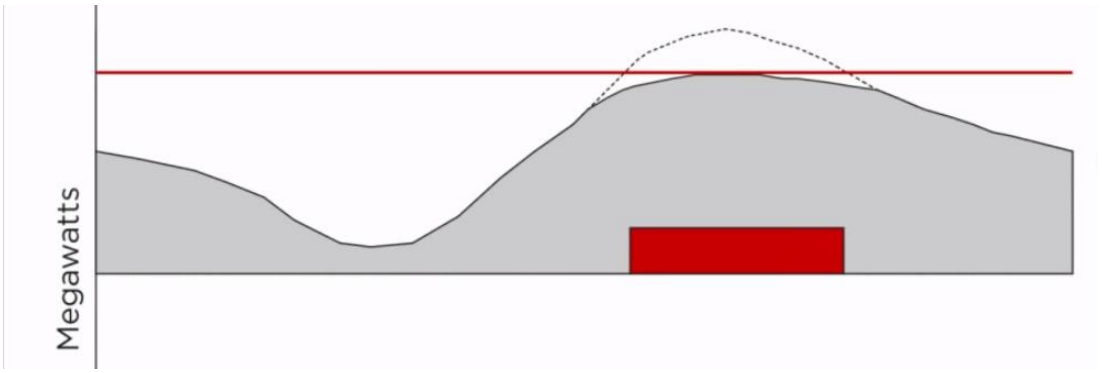
Before we see the software architecture for ***peak shaving***, let us describe the problem that this tries to solve



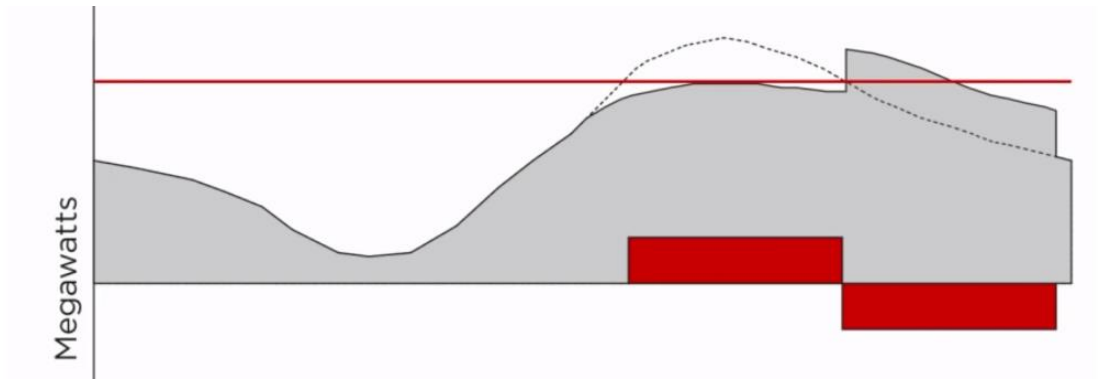
This is a ***graph of aggregate grid load profile in MW*** for a warm summer day, the grid load varies with the weather and the time of the year



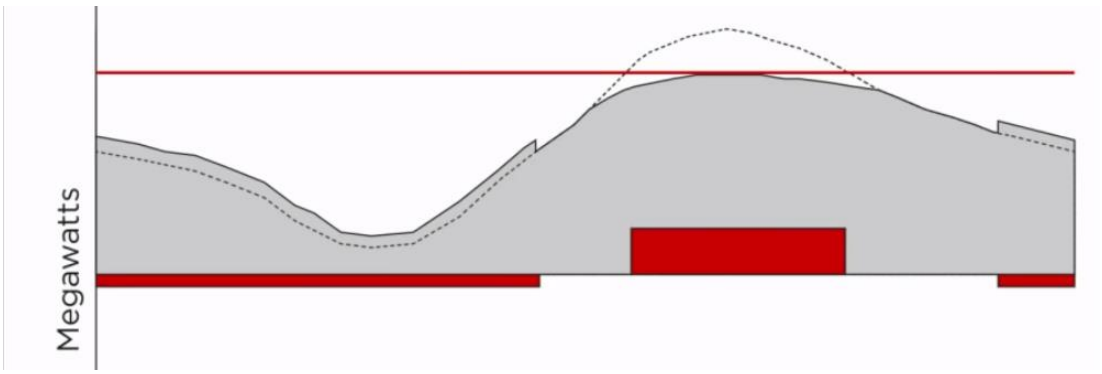
Peak loads are very expensive and the grid only needs to meet the peak load a few hours in a year. The 2 options for a utility to satisfy peak loads are to build more capacity via new power plants, or to import power from another utility.



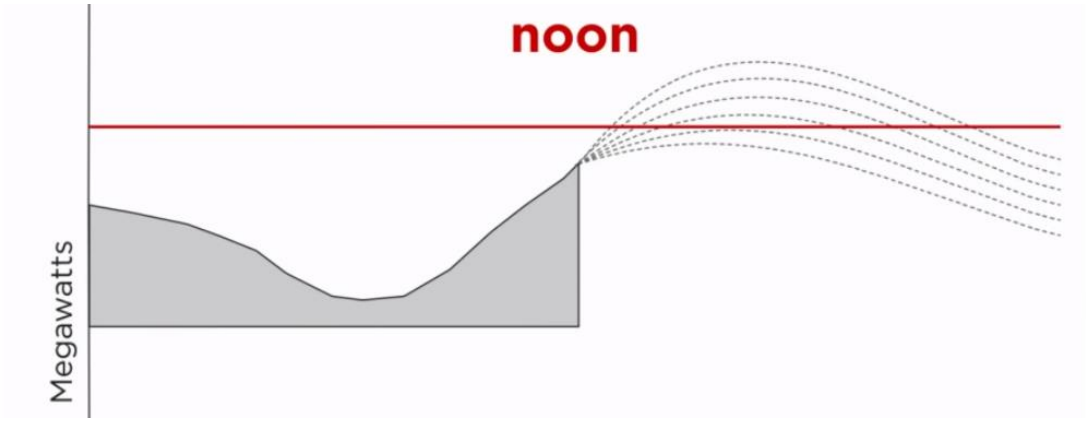
Power can be cheaper if we can offset demand and make the load curve more difficult, we want to discharge PowerWall batteries during the peak load times and let the home owner use the battery for clean backup power at other times



We learnt not to charge the battery back up immediately after a peak

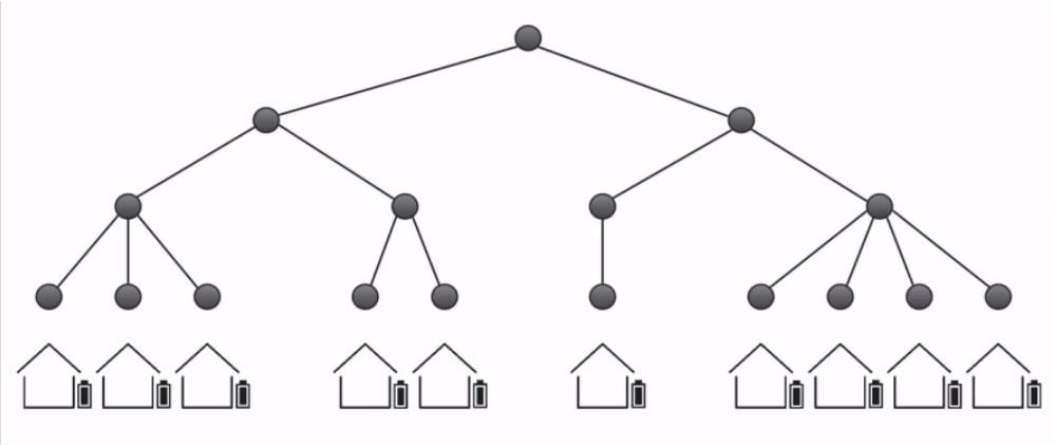


The solution is to control when to discharge and charge up the battery for maximum cost benefits. Above is the picture we are trying to accomplish

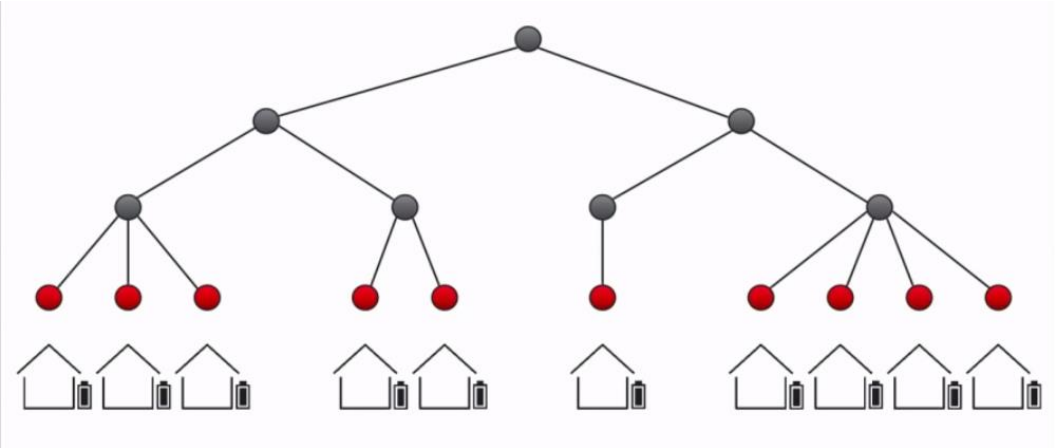


We need to be able to also predict peaks and determine if to discharge the battery or not.

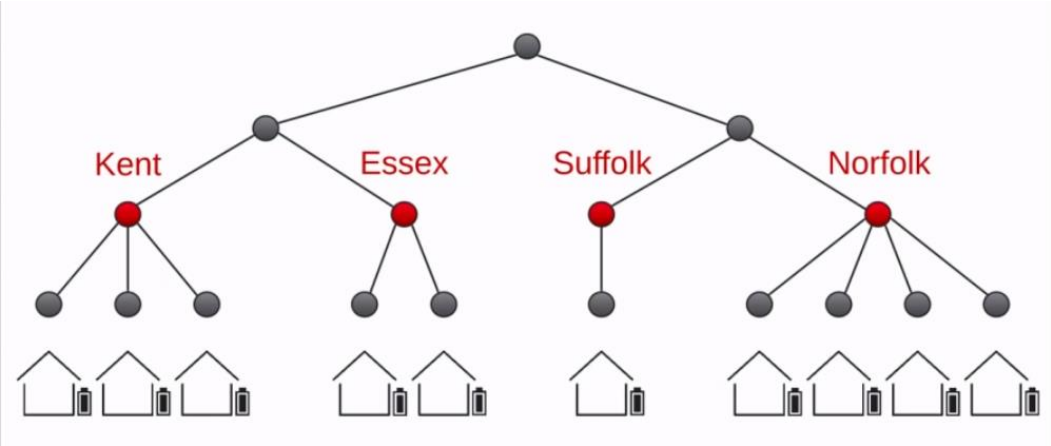
Once we have decided to discharge the battery at some peak, how do we control the participating PowerWall batteries that are enrolled in the specific VPP program in the locality?



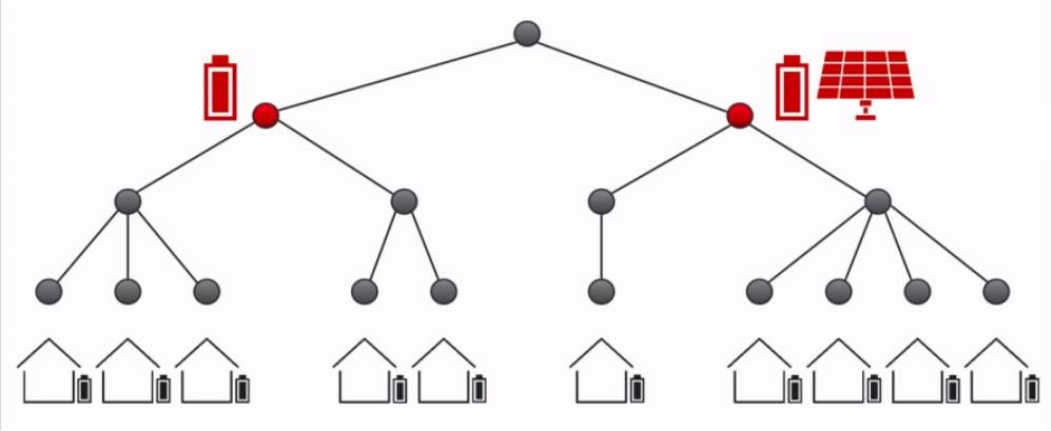
The grid is not designed to interact with a whole bunch of small power producers, we need a way to aggregate all the power and deliver to the grid efficiently using hierarchical software representations



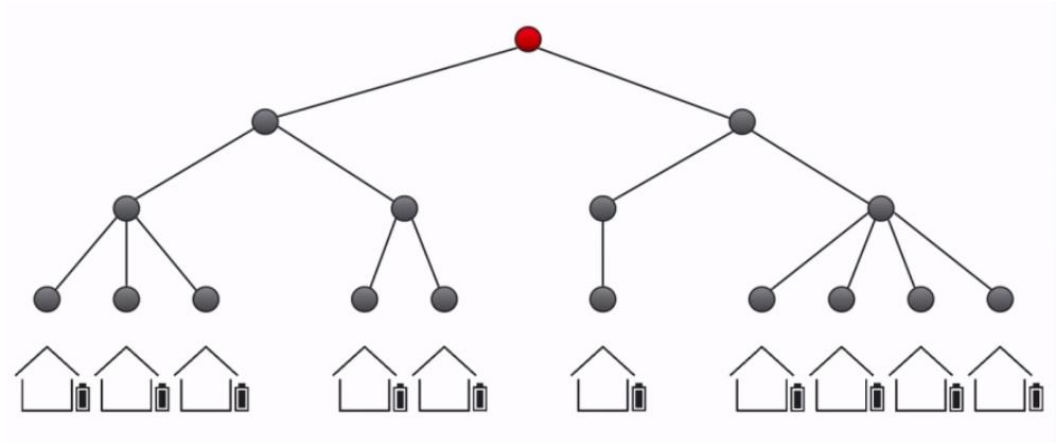
The first level is the digital twin that represents an individual site, i.e a house with a PowerWall



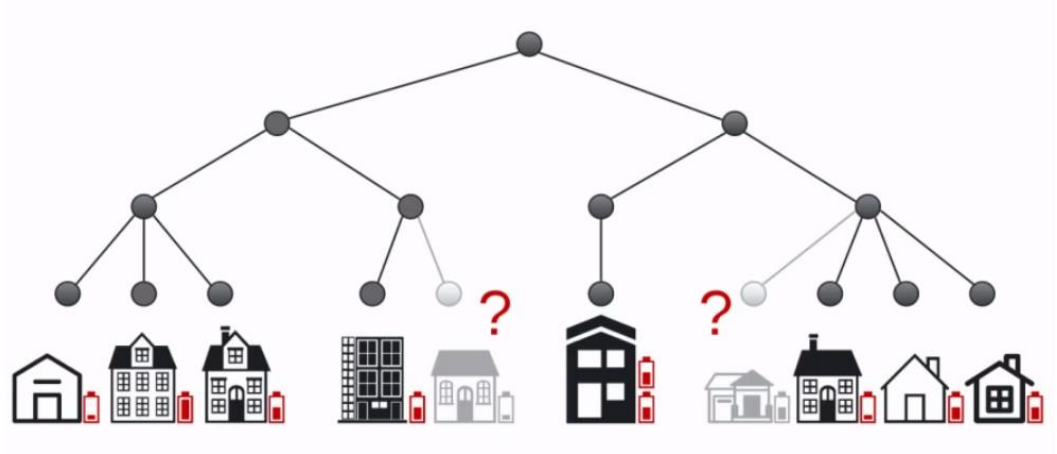
The next level is organized by electrical topology like a substation or by geography like a county



Then we have a physical grouping like an electrical interconnection or logical like sites with a battery and solar that we want to optimize differently.

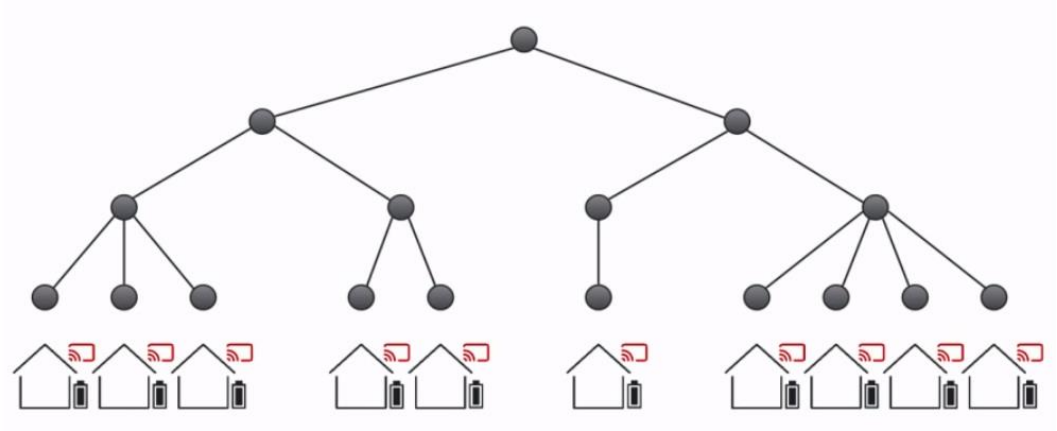


These all come together to form the top level of the VPP where we can query the aggregate of thousands of PowerWalls quickly enough to rival a single physical power generating plant. This aggregate is then used to inform our global optimization algorithms and bidding process.

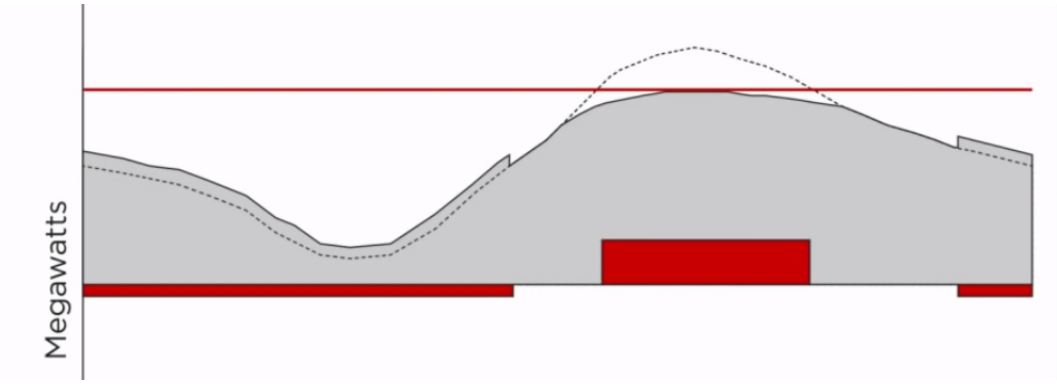


The VPP is not uniform, there is a diversity of homes and the available PowerWall count, sizes, configuration, sunlight charge variations, communication availability and usage patterns that go into the optimization.

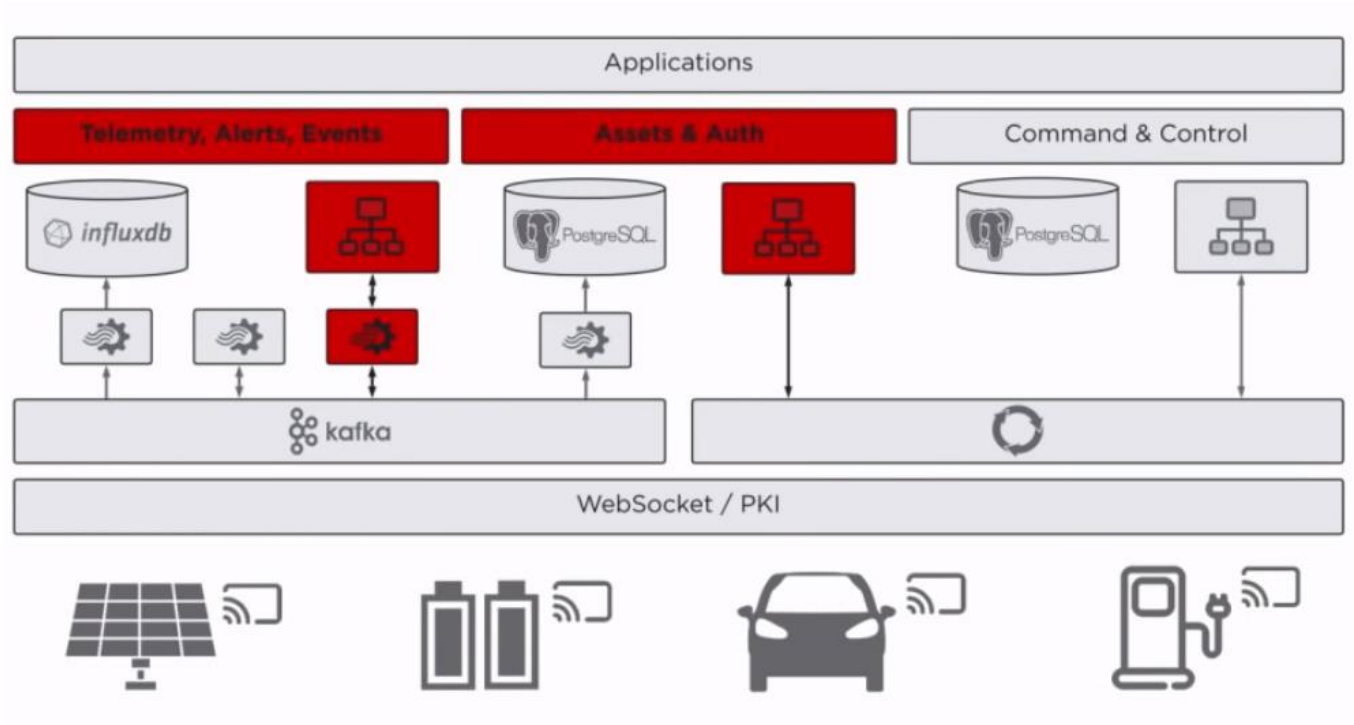
We have to represent the uncertainties in the data model and the business logic. We want to say things based on the local context like ‘there is 10MW/hrs of energy available, but only 95% of the sites we expect to be reporting have reported’.



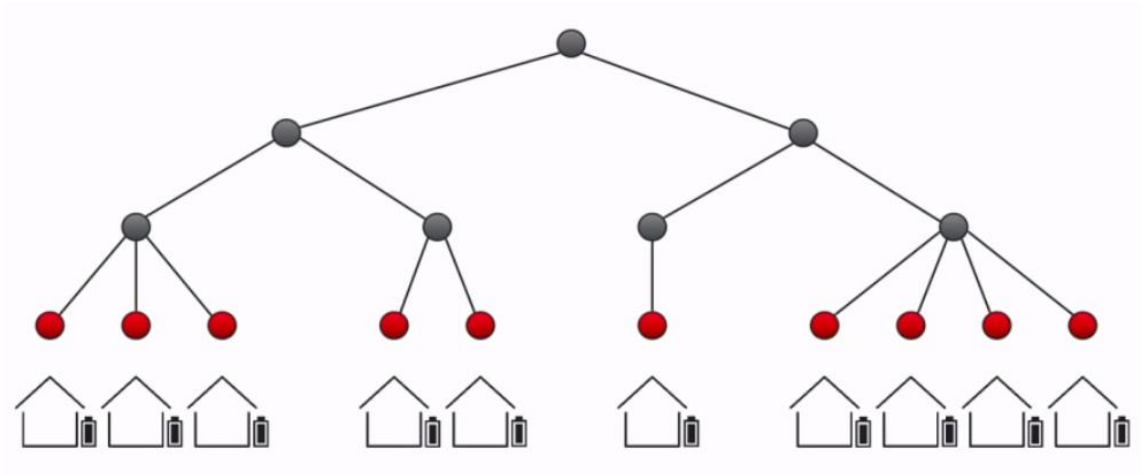
One way we manage this uncertainty is through a site level abstraction despite the heterogeneity via an edge computing platform that provides site telemetry for things like power, frequency, voltage that gives us a consistent software abstraction



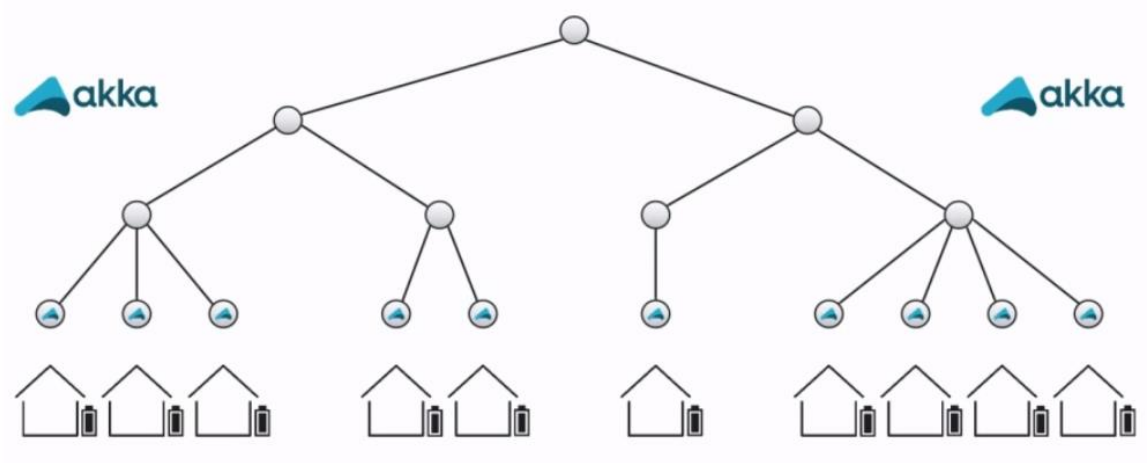
Another way is to aggregate the telemetry across the VPP to allow people/operators/algorithms not worry about individual power plants and instead focus on how to discharge 10MW from 5pm to 6pm on a given day.



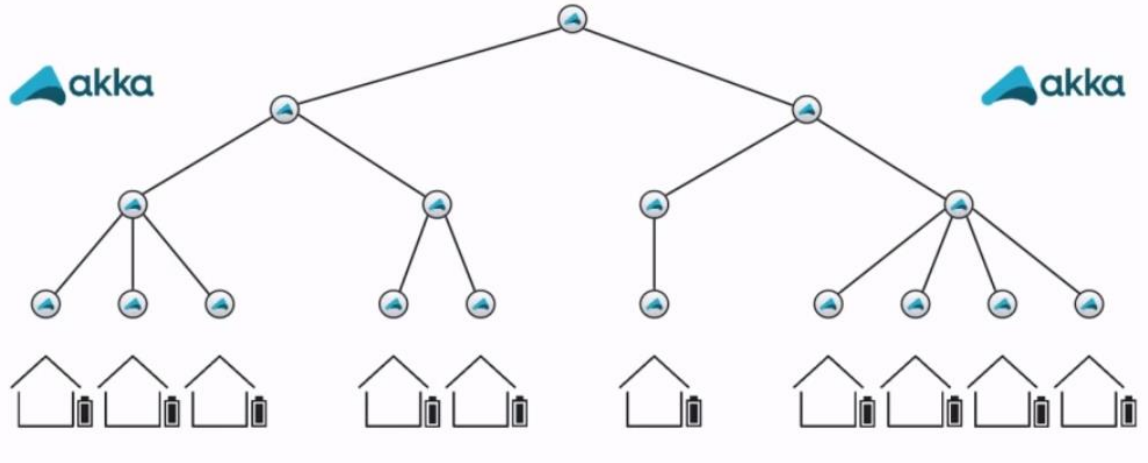
This is a difficult engineering challenge that is a combination of streaming telemetry and asset modelling



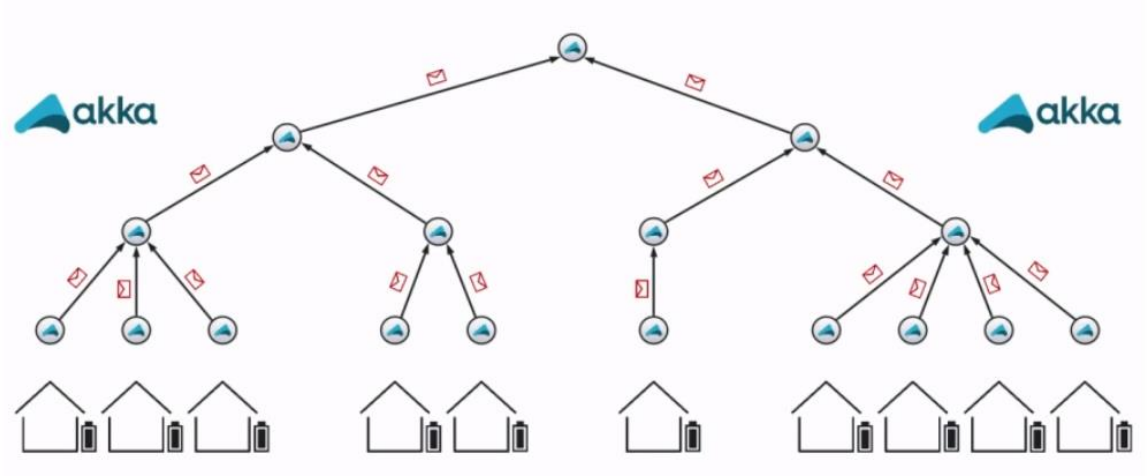
We use the digital twin for modelling each site in software,



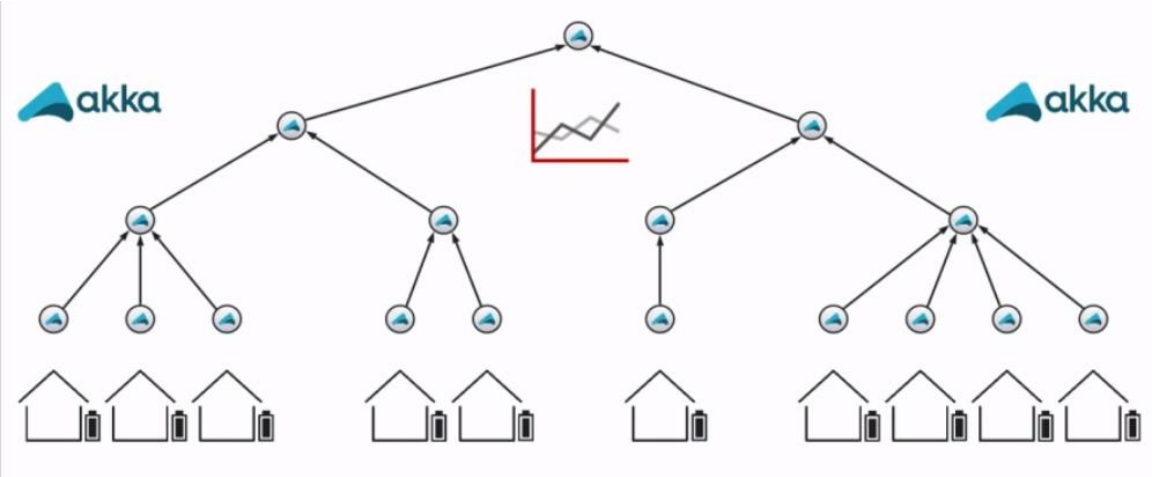
We represent each site with an Akka Actor that manages state like the latest telemetry reported from that battery, it executes the state machine that changes its behavior if the site being offline and the telemetry is delayed. It also provides a convenient model for concurrency and computation. So, the programmer worries about modelling an individual site in an actor, then the actor runtime handles scaling this to thousands of sites. This is a very powerful abstraction for IoT use cases that allows us not to worry about threads, locks or concurrency issues.



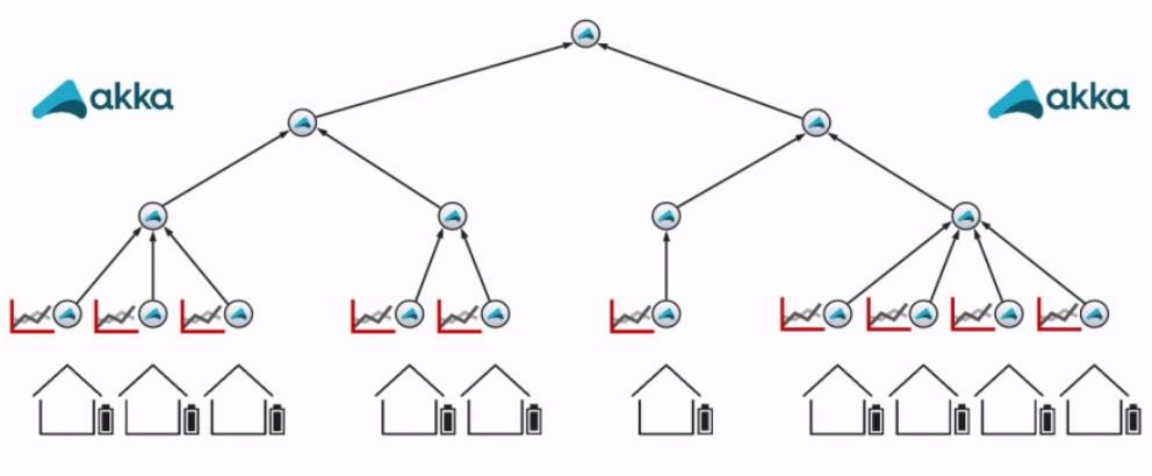
The higher-level aggregations are also represented by individual actors, the actors maintain their relationships with other actors to describe the entire physical, logical aggregations.



The telemetry is aggregated by messaging up the hierarchy in-memory and in near real-time. How rea-time an aggregate is at any level is a tradeoff between messaging volume and latency.

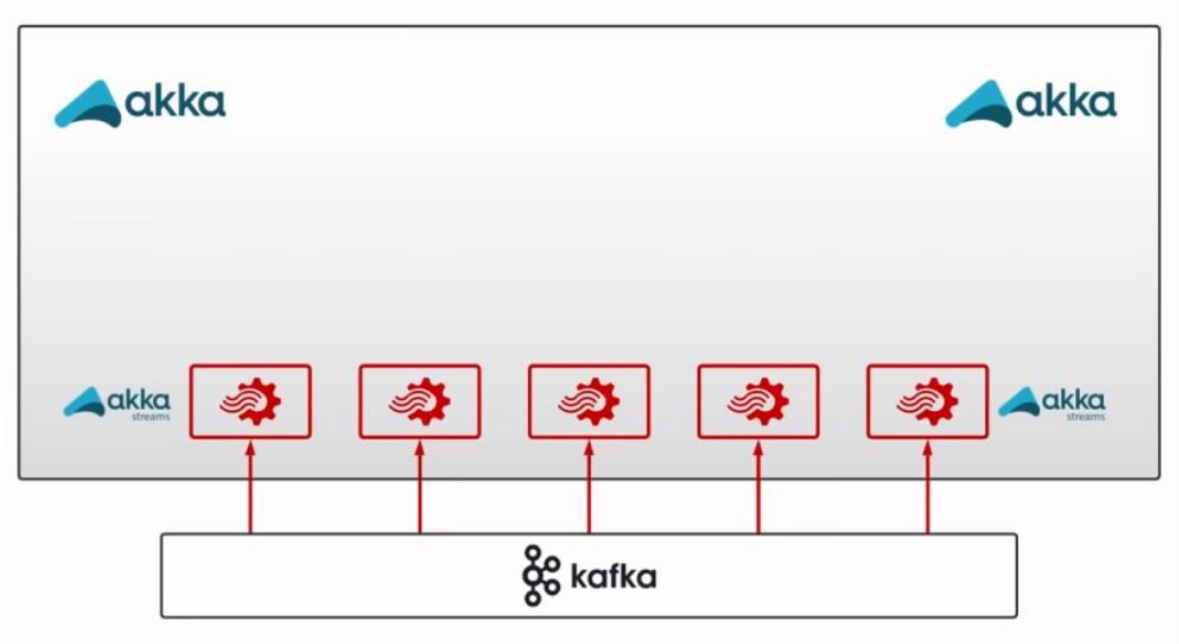


We can query at any node in this hierarchy to know the aggregate value at that location

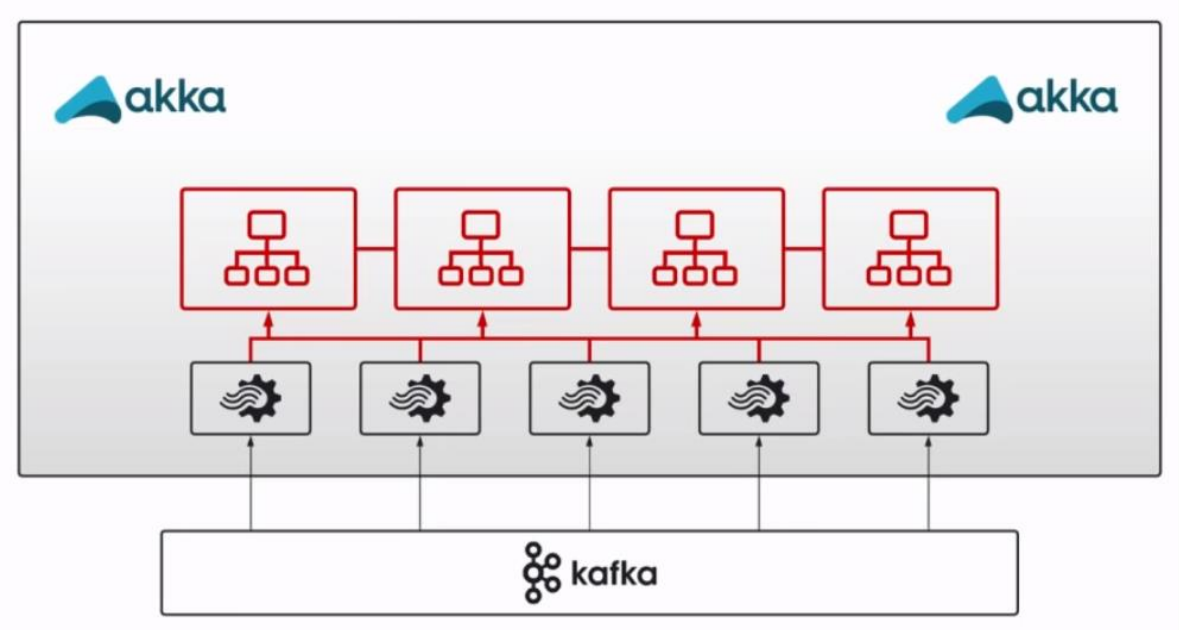


We can also query the latest telemetry from an individual site, we can also navigate up and down the hierarchy from any point

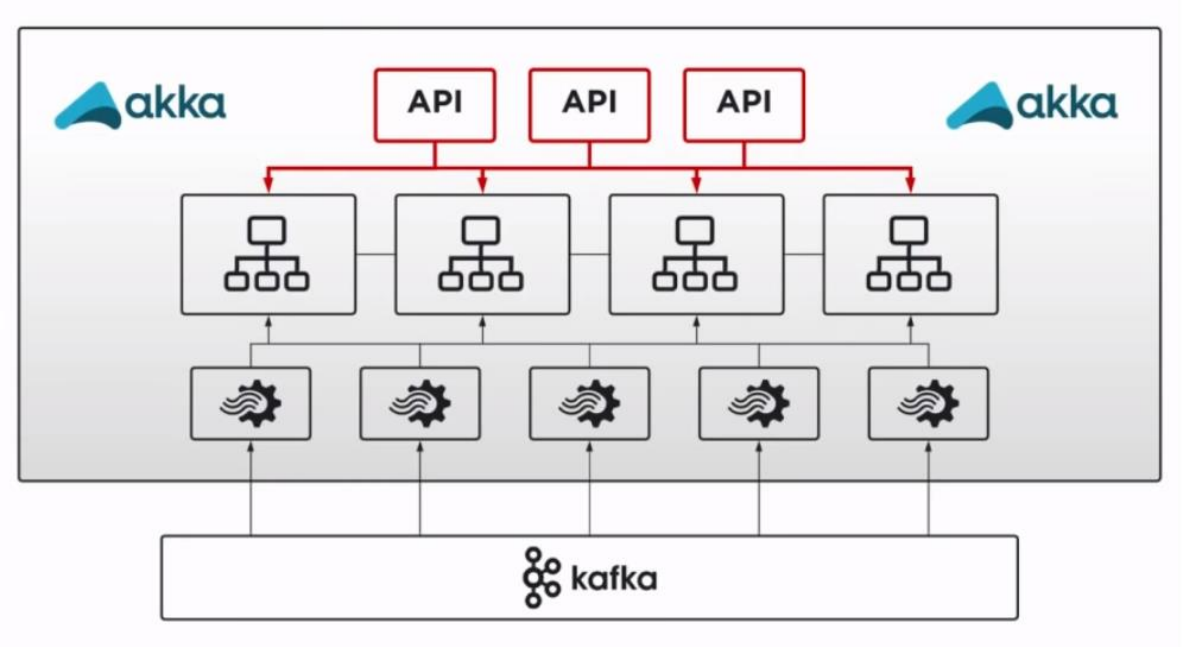
The services that perform this real time hierarchical aggregation functionality in real time run in an Akka cluster. An Akka cluster allows a set of pods with different roles to communicate with each other transparently



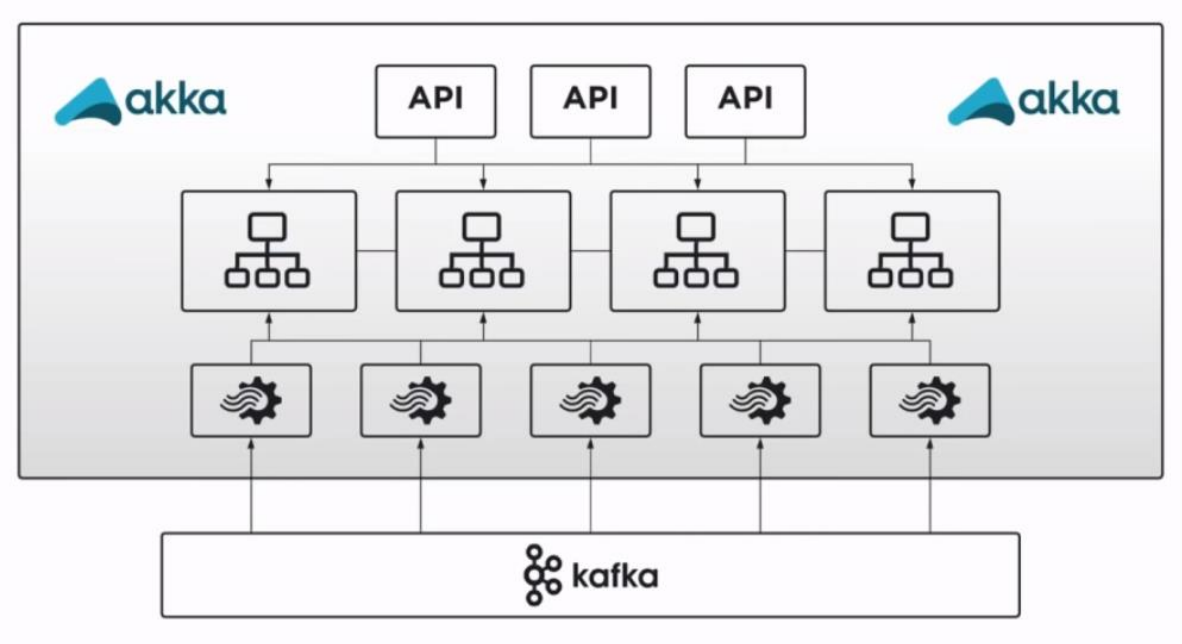
The first row is a set of linearly scalable pods that stream data off Kafka, they use Akka streams for back pressure, bounded resources constraints, low latency stream processing.



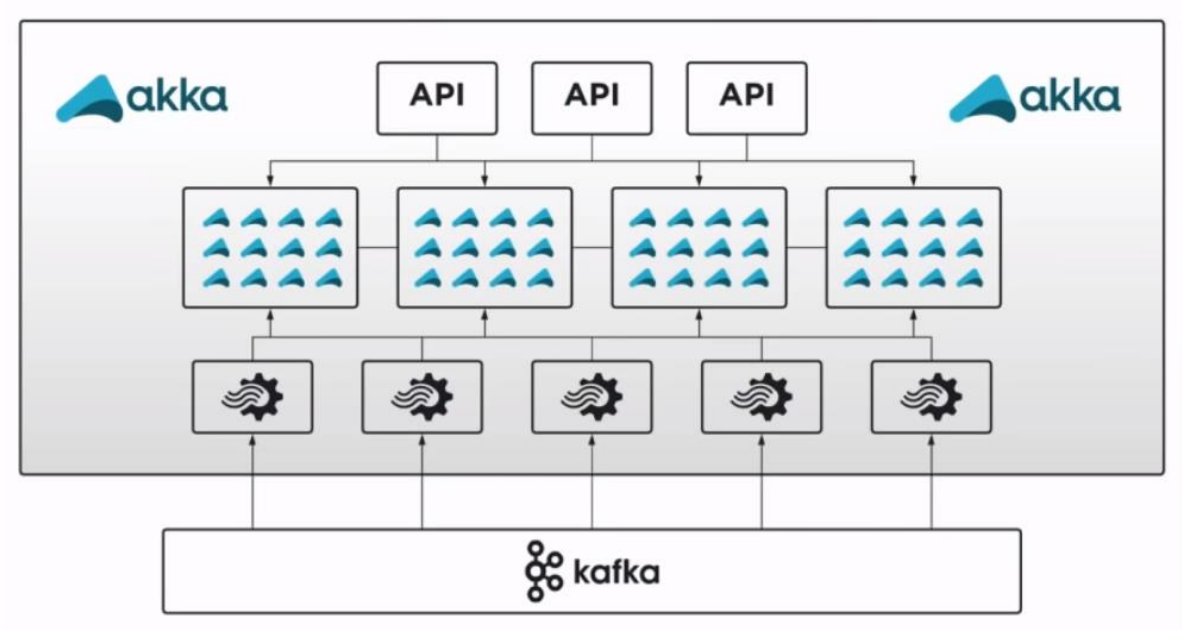
They message with another set of pods running all the actors in the various virtual representations like a Site. When the stream processor reads a message off Kafka for a particular site, the message to the actor representing that site simply uses the **siteId** or site identifier. It doesn't matter where in the cluster that actor is running from, the Akka runtime will transparently handle the message delivery using its location transparency feature. Site actors message with their parents in a similar manner using the parent's ID up the hierarchy.



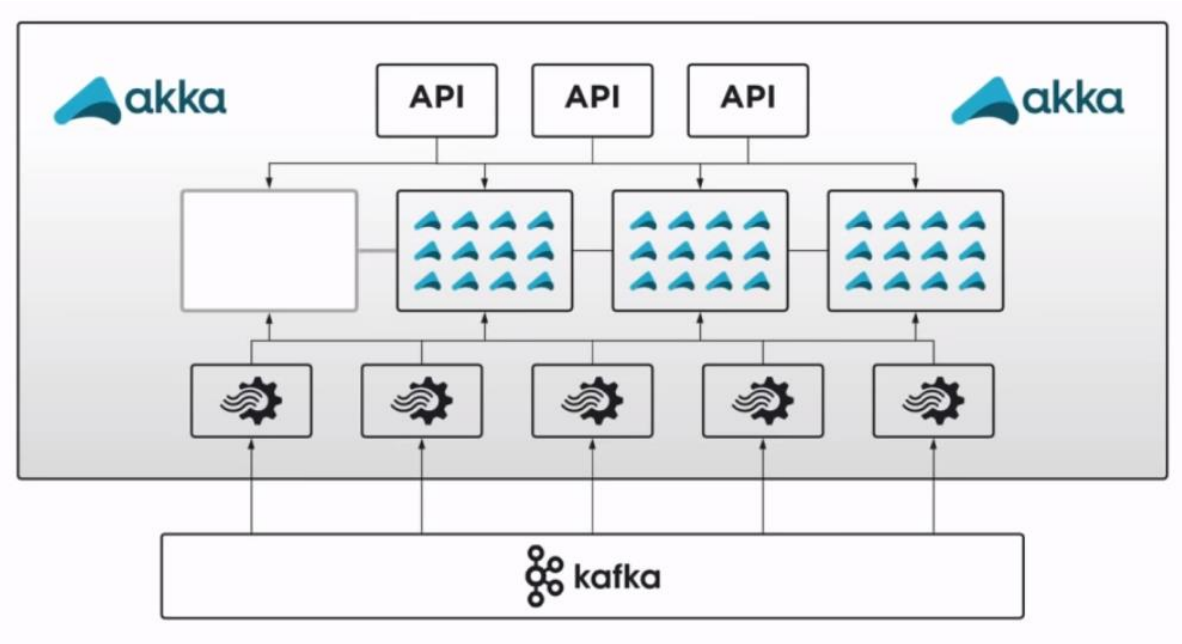
There is a set of API pods that can serve client requests for Site level or aggregate telemetry because they can query into the cluster in the same location transparent way.



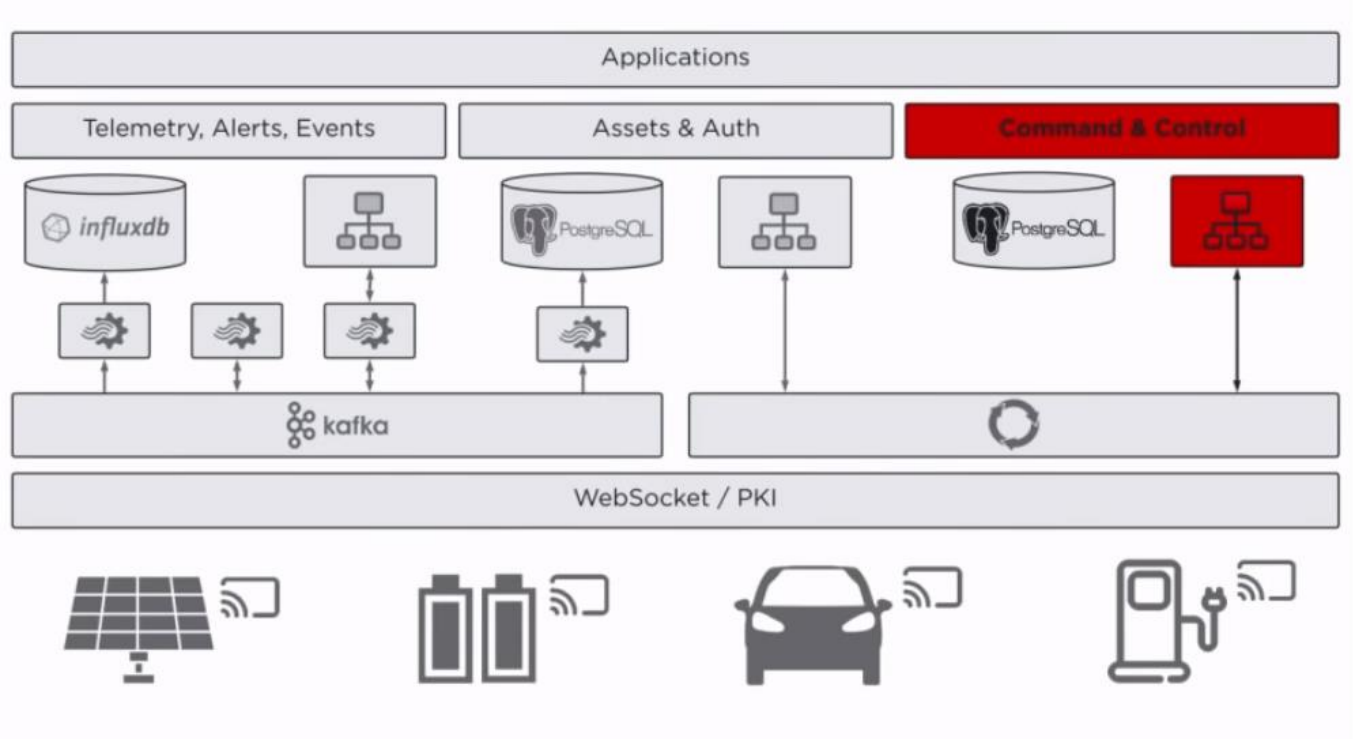
This collection of services provides the in-memory, near real-time aggregated telemetry for thousands of PowerWalls.



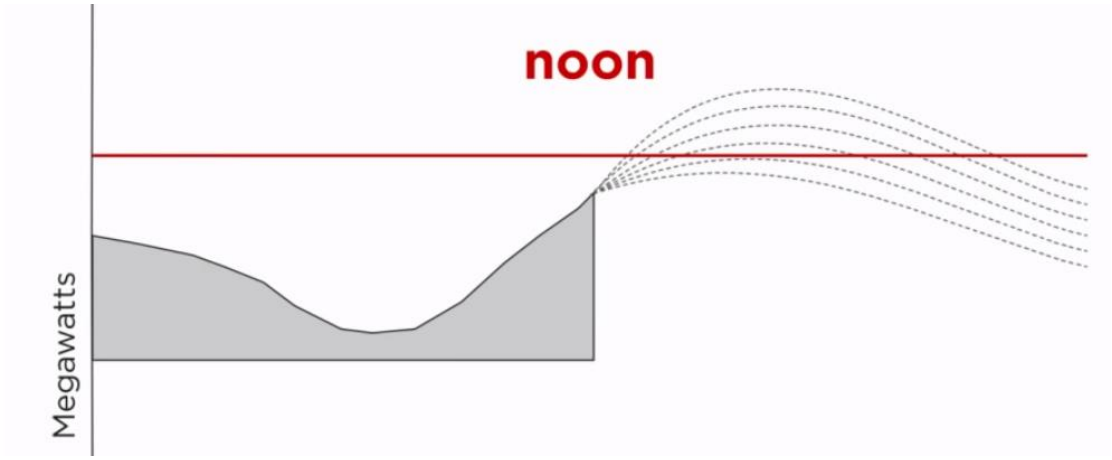
This architecture provides great flexibility, especially when paired with K8s to manage the Pods. The Actors are just running on their own compute substrate/heap



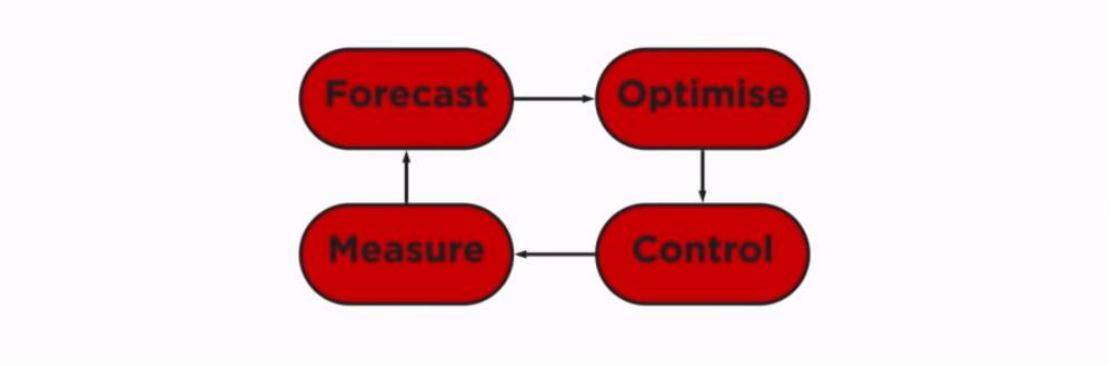
An individual actor can fail or be restarted and the actors/work simply migrates to another pod until the substrate/heap recovers. The cluster can also be scaled up or down and the actors will rebalance across the cluster. Actors can recover their state automatically by using Akka persistence but in our case, we let the actor simply rediscover its relationship and the latest state when the next message from battery arrives within a few seconds.



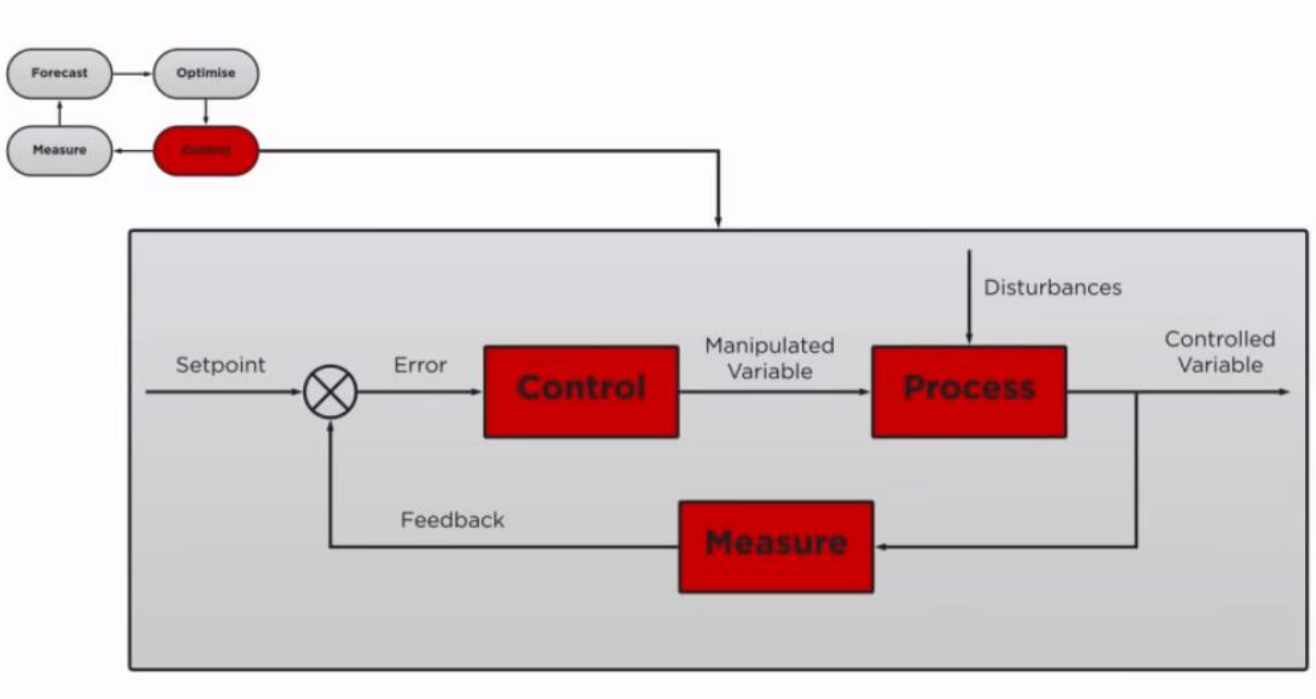
After aggregating the telemetry to know the capacity that is available from all participating battery VPPs, we then have to control the batteries.



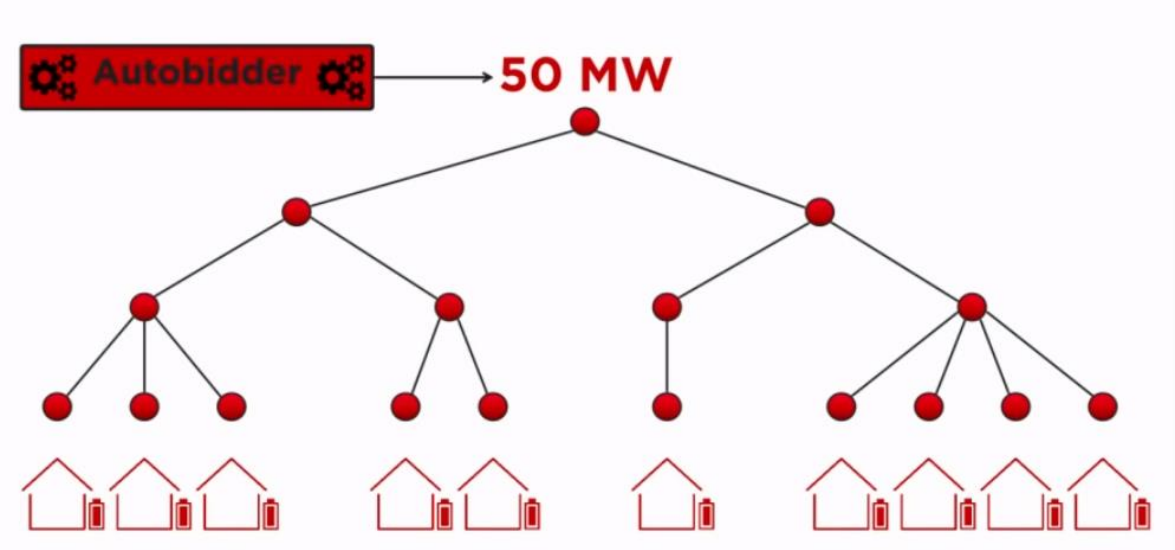
The first step is **taking past measurements**, **forecasting**, and **deciding how many megawatts to discharge** if we are going to hit a peak.



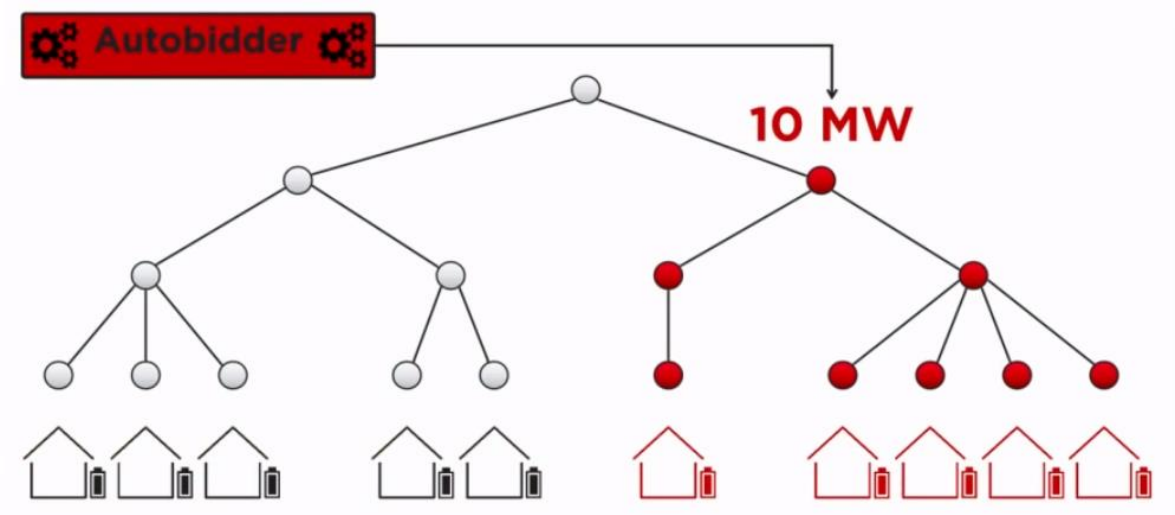
This is the loop at a high level that gets run continuously



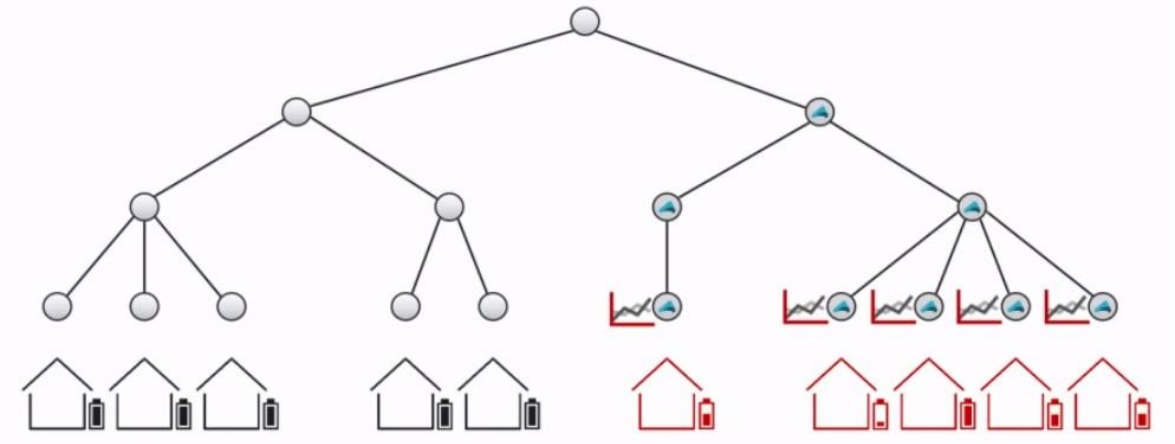
The control part of the architecture is a closed loop control as above. Once an aggregate control setpoint as being determined, we continuously monitor the dis-aggregate telemetry from all the sites to see how it responds, we the adjust the setpoints for the individual sites to minimize the error



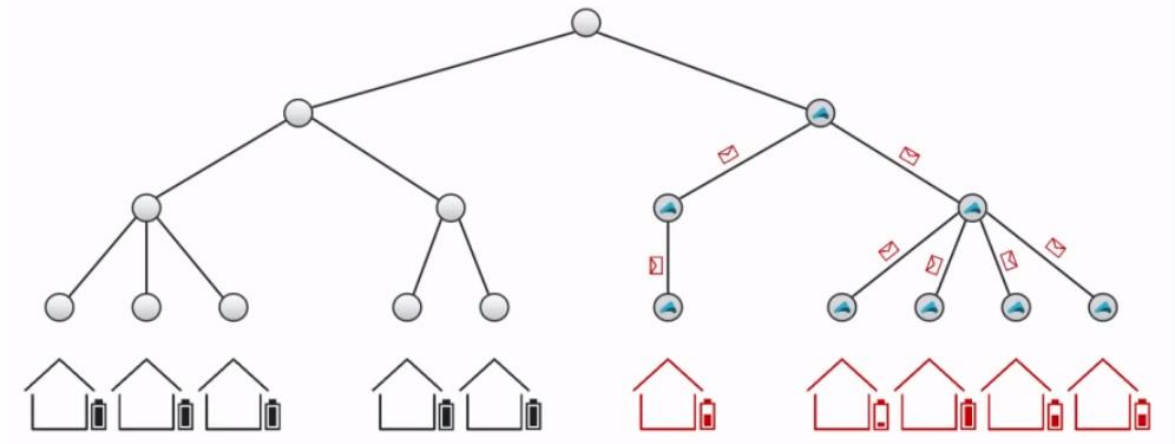
The **Autobidder** platform may decide to control the whole fleet by providing 50MW to offset the peak load shortage,



The Autobidder might instead provide 10MW by controlling a subset Sites depending on the objective. The control system/service dynamically resolves the individual sites under the Autobidder's target subset by query the assets directly.



The control system/service query's the battery telemetry at every site using the in-memory aggregation discussed earlier to decide how to discharge each of the batteries like a database query planner. E.g there is no point in discharging a battery that is almost empty.



The control service then sends a message to each site with a discharge setpoint and a timeframe, it will keep retrying until it gets an acknowledgement from the site or the timeframe elapses. Because the size of these streaming to thousands of sites can be very large, we stream over Akka streams to provide bounded resource constraints for things like resolving all the sites, reading off all the telemetry, and then sending back all the control setpoints.

Takeaways

Huge aggregations demand different APIs
and data-processing patterns.

You cannot just use typical CRUD microservices for this, you need streaming semantics for processing large collections with low latency and bounded resource constraints. You need a runtime for modelling stateful entities that supports location transparency, concurrency, scaling and resilience.

Embrace uncertainty.
Model reality.

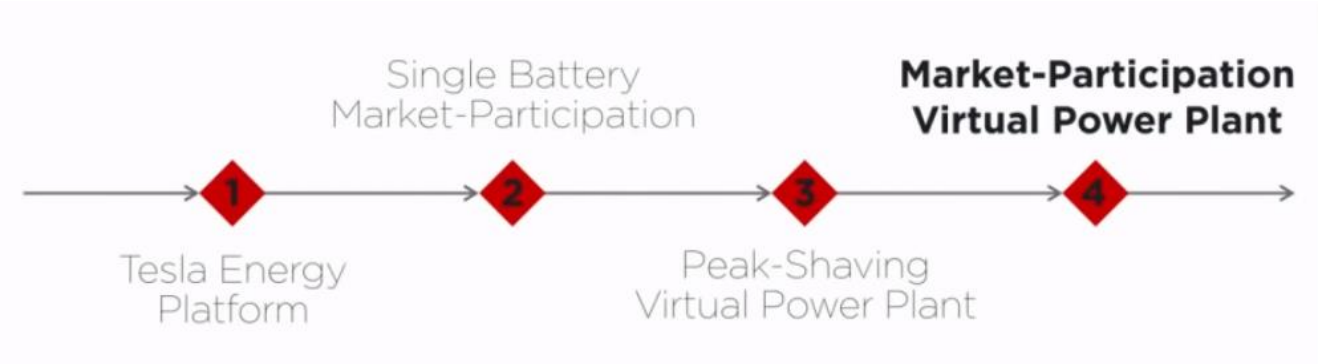
Uncertainty is inherent in distributed IoT systems so we need to embrace uncertainty in the data model, business logic, and even in the customer experience.

Asset management is the hardest problem in IoT.

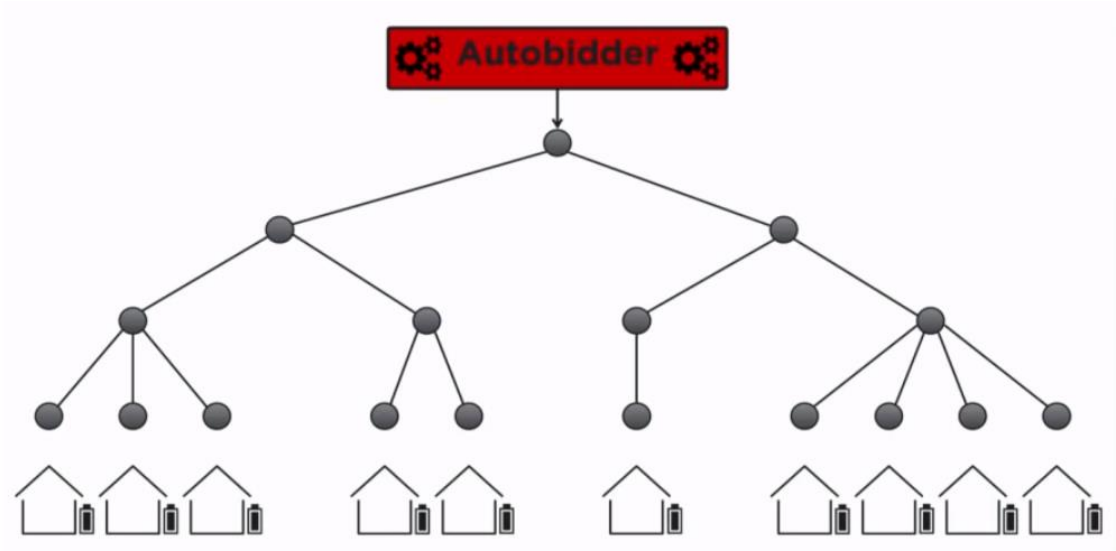
Representing physical and virtual relationships among IoT devices especially as they change over time is the hardest problem in IoT but needed for creating a great product.

There is a tension between
central optimization and local needs.

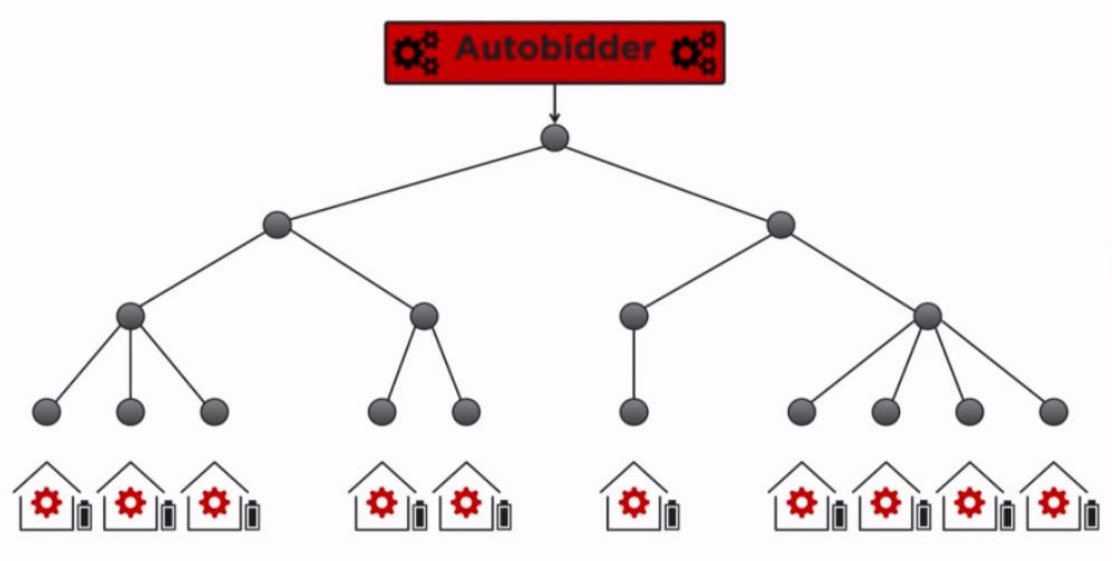
Imagine a storm is approaching close to a peak, the local objective wants to discharge the battery to avoid the peak and make some money but the home owner also wants a full battery in case the power goes out. This tension leads to the last part of our work so far, the **co-optimized virtual power plant**.



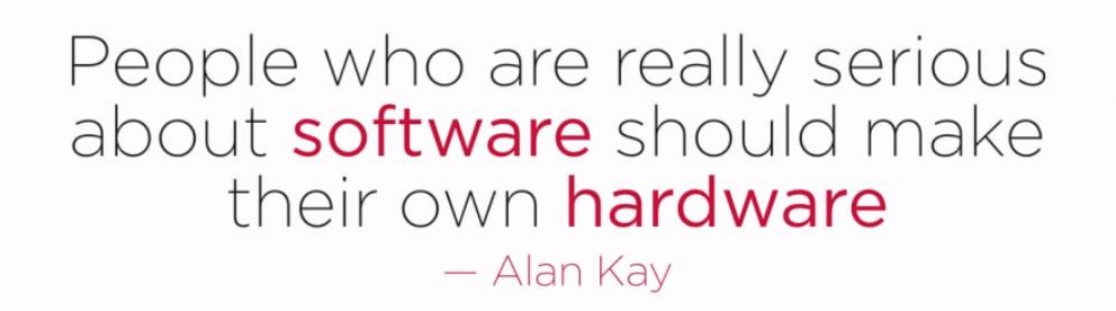
We are now going to aggregate and optimized thousands of batteries, but this time not just for a global goal but to co-optimize for local objectives as well.



At the peak-shaving VPP we described earlier that optimizes for a central objective and passing the control decisions down to all the participating sites.

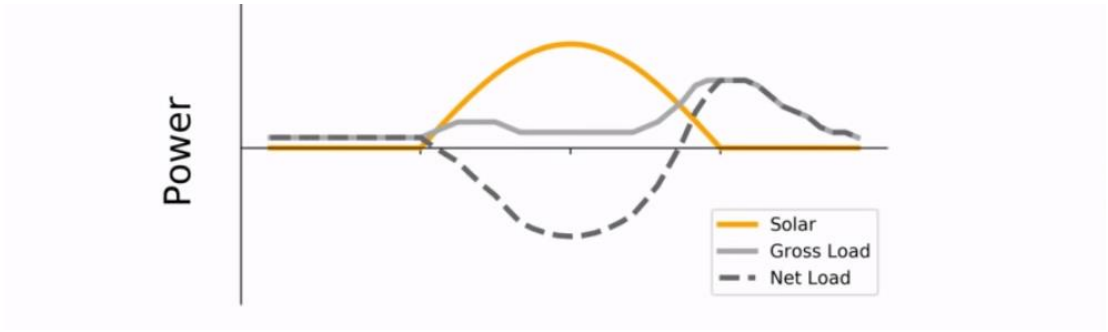


The optimization decisions are distributed across the sites but the sites now participate in the control decisions.

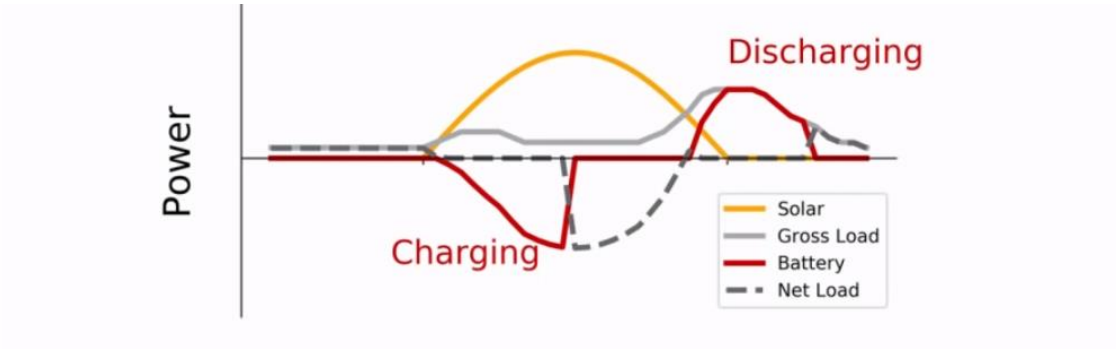


This distributed control is only possible because Tesla built and has full control over its own hardware and software. This enables quick decisions and iterations across the central intelligence and how they relate to each other.

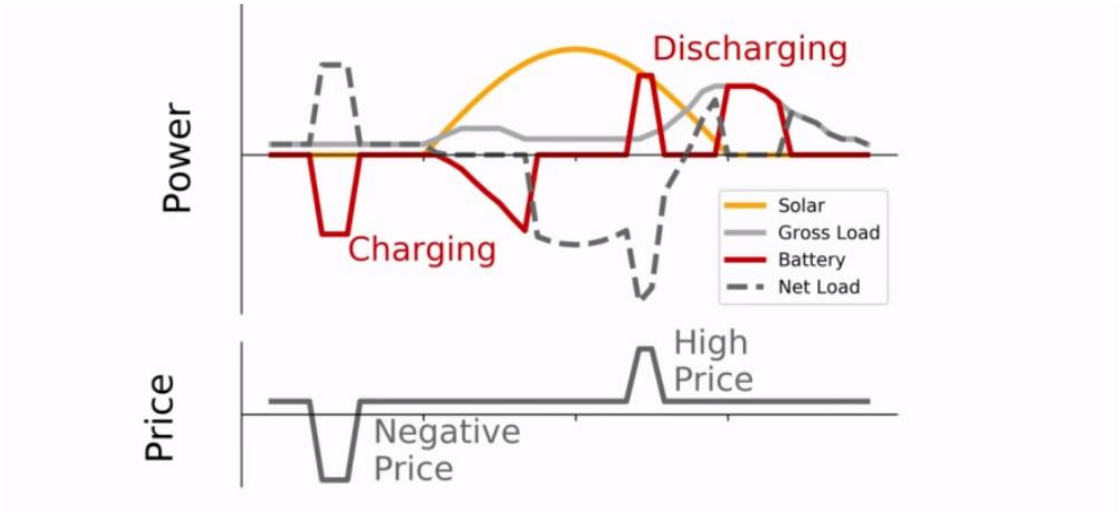
What do we mean that the VPP co-optimizes global and local objectives?



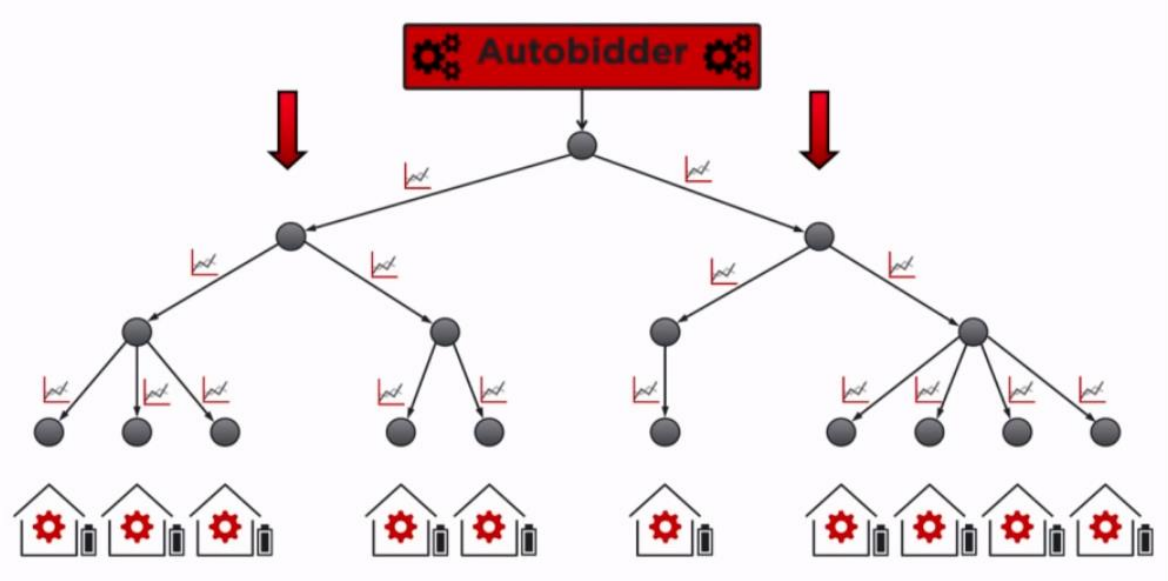
The net load needed is what the utility sees and tries to optimize for.



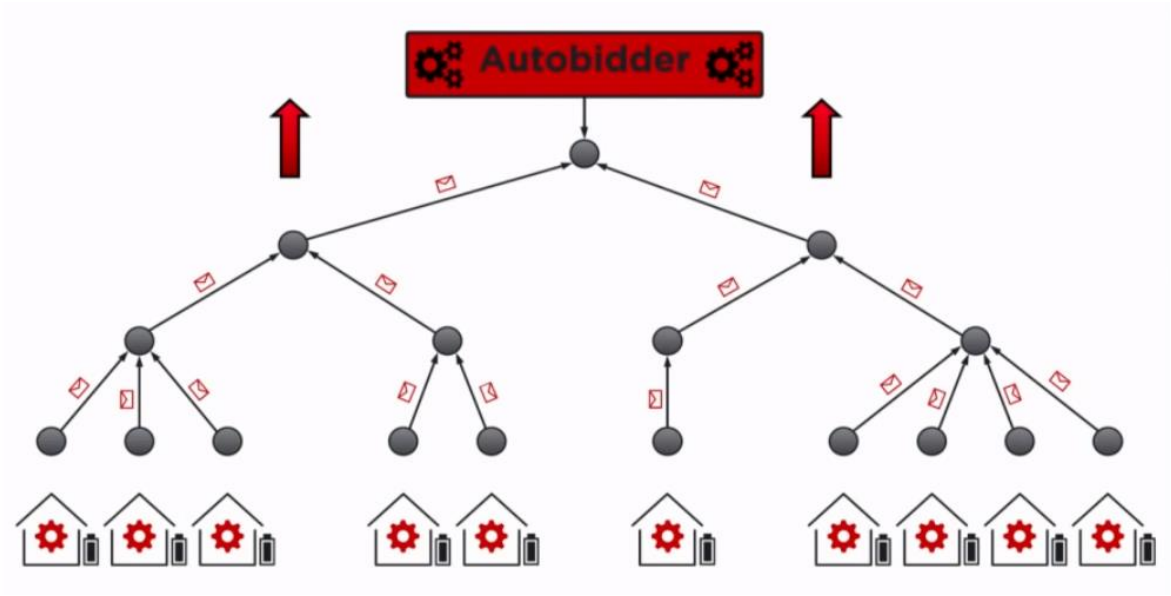
Since the PowerWall battery can charge up during excess loads and discharge during peaks using the local intelligence on the device (for maximizing the solar power being used or minimizing the user's utility bill), this is local optimization.



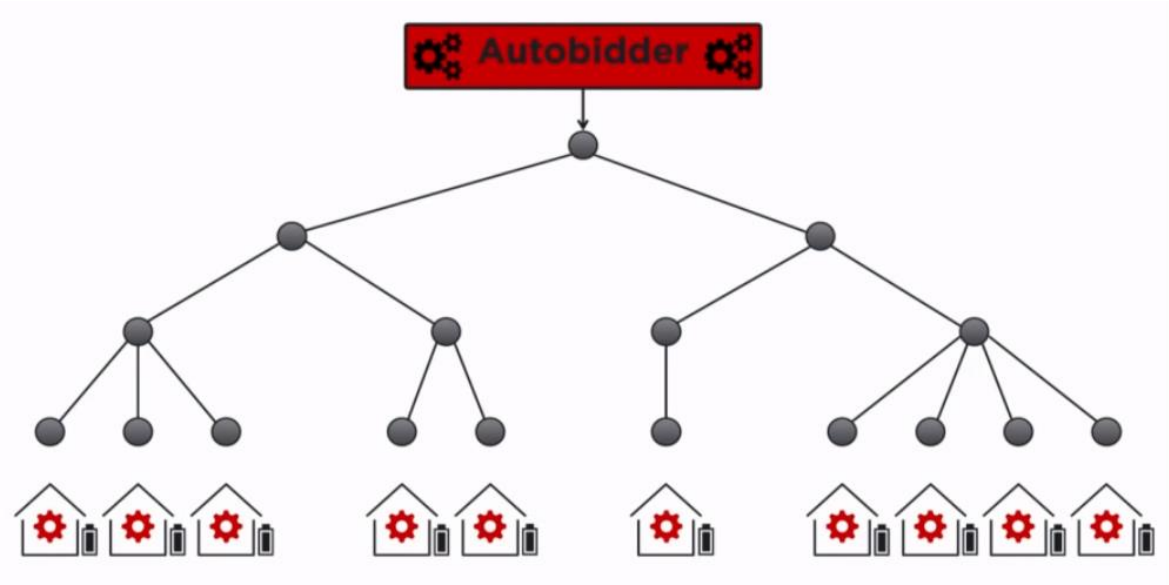
We can co-optimize the local and global objectives as above. We can also consider information about the aggregate goal like market prices that indicate the real time balancing needs of the grid. Negative pricing in the night caused by wind generation into the grid can cause the battery to take power from the grid and also get paid for it. A high price in the afternoon due to high demand can cause the battery to discharge rather than waiting to be fully charged.



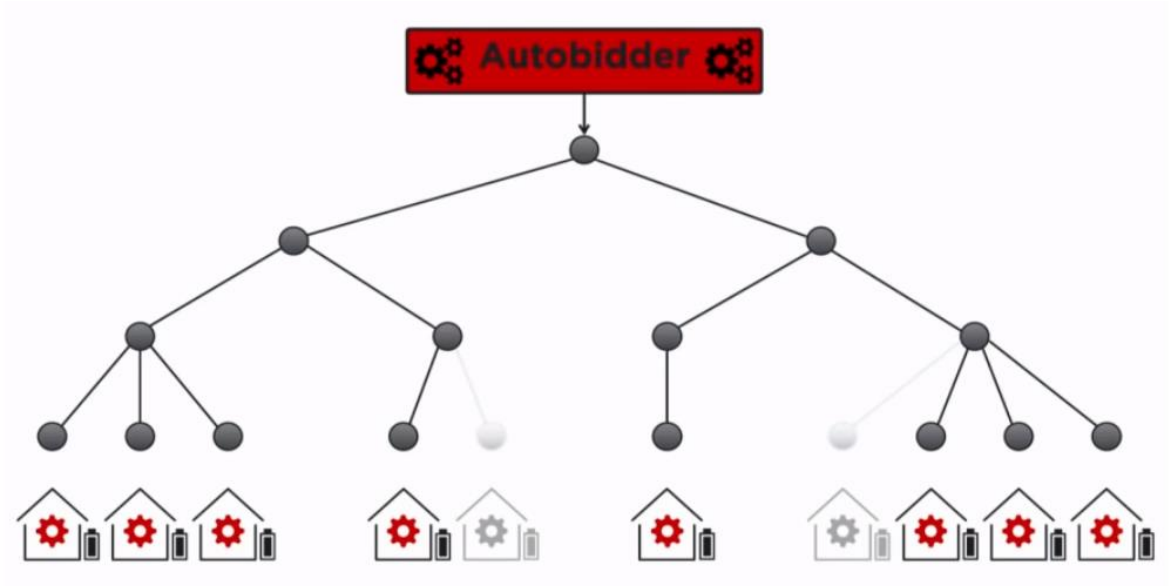
In our co-optimized VPP, Autobidder generates a time series of price forecasts every 15 minutes. The Tesla Energy Platform control components distributes those forecasts to the sites.



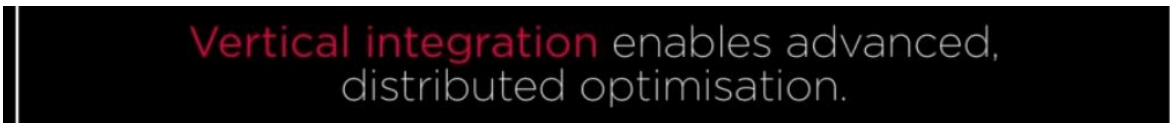
The local optimization then runs, makes a plan for the battery using both the local and global objectives, then communicates the plan back to the TEP for ingestion and aggregation using the same ingestion framework for telemetry. Autobidder then uses the aggregate plans to decide what to bid for on the energy market.



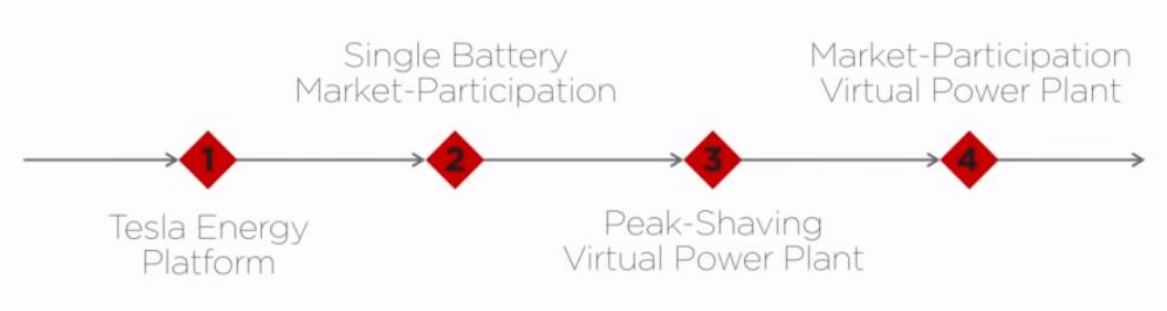
This distributed architecture takes advantage of the edge computing power available on the PowerWalls, it also does not solve a large, single optimization problem over all the sites. New sites can join the aggregation without affecting the central optimization.



Another advantage is resilience for when sites go offline or communication is lost over short time periods, we can continue to co-optimize using the last data point and estimates.



Algorithms depend on the software platform.



Proposals in the grid space typically discuss "algorithms"...

— Astrid Atkinson

But the key part of modern distributed computing systems is the system itself.

— Astrid Atkinson

Software is key to a clean energy future.

TESLA