

PROMPT

Below is a revised version of the research plan, updated to reflect the key constraint that no additional software can be installed in the Kubernetes pods—only a bare-bones Python interpreter is available. All intermediary or caching functionality must therefore reside outside the pods (in a separate microservice, sidecar, or an external service), and any communication from the pods must use standard Python libraries (e.g., `urllib`, `http.client`) that require no extra installations.

1. Define the Problem and Objectives

Problem Statement

- Kubernetes pods submit jobs to a Grid Engine, then directly poll the Grid Engine's queuemaster for job status.
- The high polling frequency overwhelms the queuemaster, delaying new job dispatch and causing unwanted autoscaling.

Critical Constraint

- No additional software can be installed in the Kubernetes pods. Each pod only has a bare-bones Python interpreter (standard library). This excludes installing external Python libraries (e.g., `requests`, `websocket-client`) or system utilities (e.g., `curl`, `wget`).

Objectives

1. Offload frequent polling from pods to an intermediary service or architecture that:
 - Reduces load on the queuemaster.
 - Can be built using any technology outside of the pods.
2. Minimize code changes inside the pods:
 - Pods must rely on standard library Python calls to query job statuses.
 - No additional binaries or libraries can be installed in the container image.
3. Improve overall system responsiveness and prevent autoscaling churn in Kubernetes.

2. Background and Literature Review

Topics to Review

- **Polling vs. Event-Driven Architectures** Investigate how systems transition from heavy polling to push notifications, webhooks, or broker-based event streams.
- **Caching and Aggregation Patterns** Explore ways to maintain a near-real-time job status cache, ensuring pods query only this cache rather than the queuemaster.
- **Relevant Job Scheduler Insights** Study how other schedulers (e.g., Slurm, Univa) decouple status monitoring from scheduling to handle large-scale polling.

Sources to Consult

- Kubernetes Documentation on Jobs, controllers, and how to reduce unnecessary pod restarts or autoscaling triggers.
- Grid Engine Guides for best practices in handling high-frequency queries or external integration.
- Distributed Systems References on caching, message-driven designs, and external aggregator services.

3. Identify Candidate Intermediary Approaches

In all approaches below, the pods remain untouched beyond a simple Python script that makes standard-library HTTP requests (e.g., via `urllib.request` or `http.client`). All new functionality (caching, event handling, etc.) resides in an external microservice or service layer.

Approach A: Caching/Proxy Microservice

1. **Design**
 - Build a lightweight, scalable microservice that polls the Grid Engine at a sensible interval (e.g., every few seconds).
 - It caches or stores the job status results in memory or in a small key-value store.
 - Pods make a single HTTP call to this microservice (using Python's standard `http.client` or `urllib`) to retrieve job status.
2. **Technologies**
 - External microservice can be written in Go, Node.js, Python (with a full environment), etc.
 - Cache layer could be in-memory or use Redis.
 - No changes to the pods (other than pointing them to the microservice endpoint).
3. **Pros/Cons**
 - **Pros**
 - Offloads the polling load from the queuemaster to the microservice.
 - Easy to scale the microservice separately from pods.
 - Requires no changes to Grid Engine internals.
 - **Cons**
 - Need clear cache invalidation/refresh logic to ensure status freshness.
 - The microservice could become a single point of failure if not designed for HA/replication.

Approach B: Event-Driven / Push Notifications

1. **Design**
 - Integrate (or extend) the Grid Engine so that job status changes are pushed to an external message broker (Kafka, RabbitMQ, or similar).
 - A separate microservice subscribes to these events and maintains a real-time job status cache.
 - Pods

request status via a simple HTTP call (again, using Python's standard library) to the microservice. 2. Technologies • Message queue/broker (Kafka, RabbitMQ) or a streaming system (e.g., NATS). • Microservice that consumes the job update events and exposes a REST endpoint. 3. Pros/Cons • Pros • Eliminates repeated polling to the Grid Engine (only the microservice is pushing/pulling from the broker). • Faster reaction times to job status changes. • Cons • Requires installing/configuring a message broker and hooking the Grid Engine to it (non-trivial if the Grid Engine cannot be easily modified). • Higher architectural complexity. Approach C: Hybrid Polling with Adaptive Backoff 1. Design • An intermediary microservice that offers a "long-polling" or event-based endpoint. • Pods issue requests with standard Python libraries to the microservice. The microservice either responds immediately with job status updates or instructs the client to back off if no updates are available. 2. Technologies • A custom API with an HTTP endpoint that supports either long polling or a stateful streaming approach (e.g., SSE—Server-Sent Events—but that may need additional libraries on the microservice side). • The pods themselves can do basic while-loops in Python with backoff intervals if no update is received. 3. Pros/Cons • Pros • Reduces polling frequency under steady state while delivering near-instant updates on status changes. • Minimizes load spikes on the queuemaster. • Cons • Still requires designing more complex logic in the microservice and possibly the pods (if they need to handle timeouts and retry intervals). • SSE or WebSockets in Python standard library are more limited and may not be as straightforward without installing extra packages.

RESPONSE

I will compare the three proposed approaches—Caching/Proxy Microservice, Event-Driven Push Notifications, and Hybrid Polling with Adaptive Backoff—based on implementation complexity, scalability, fault tolerance, and feasibility given the constraints of your environment. Additionally, I will find relevant case studies and potential architectures that illustrate best practices for similar implementations. I'll let you know once I have gathered this information.

Offloading Polling from Kubernetes Pods: Comparing Three Approaches

Context: In a Kubernetes cluster where pods need to query a Grid Engine (an external HPC job scheduler) for job statuses, continuously polling can be inefficient. We consider three approaches to offload or improve this polling mechanism:

1. Caching/Proxy Microservice – a dedicated service that polls the Grid Engine and caches results for pods.
2. Event-Driven Push Notifications – an event-based system where job status changes are pushed to pods (or an intermediary) instead of polled.
3. Hybrid Polling with Adaptive Backoff – pods still poll the Grid Engine but adjust polling frequency dynamically to reduce load.

We compare these approaches across key criteria: implementation difficulty, scalability, fault tolerance, latency/responsiveness, and feasibility under the given constraints (pods can only use standard Python libraries). We also highlight real-world examples, relevant design patterns, Kubernetes best practices, and recommendations for further improvement.

1. Caching/Proxy Microservice

A caching/proxy microservice involves deploying an intermediary service that communicates with the Grid Engine on behalf of pods. Pods query this service (e.g. via HTTP) for job statuses, and the service handles polling and caching of results.

- **Implementation Difficulty:** Moderate. This approach requires developing a microservice (or controller) that interfaces with the Grid Engine's API/CLI. However, it keeps pod logic simple – pods just make HTTP calls using standard libraries (e.g. urllib in Python). No additional software is needed inside pods, satisfying the constraints. The microservice can be built with common frameworks (Flask, FastAPI, etc.) or even as a lightweight Kubernetes Operator. Overall, it's easier to implement than a full event-driven system, but more work than simply tweaking polling in pods.

- **Scalability:** Good, with proper design. By centralizing queries, this service can drastically reduce load on the Grid Engine. Instead of N pods each polling, the service can poll once and serve N responses from its cache. This follows a common caching pattern: "all services will go to the same cache," improving consistency and reducing duplicate requests stackoverflow.com. It's essentially a proxy cache layer, a pattern recommended to avoid overwhelming APIs stackoverflow.com. The service itself must be scalable (it could become a bottleneck if not replicated). Kubernetes can scale this microservice horizontally behind a Service or use a distributed cache (Redis/Memcached) for consistency. Notably, Kubernetes' own API server uses a watch-cache to serve data to many clients efficiently kubernetes.io/docs/faq-architecture/– similarly, our proxy service could watch or poll the Grid Engine and serve pods from an updated cache. This approach scales well as workloads grow, provided the proxy is optimized and possibly uses caching expiration or invalidation strategies to avoid stale data.

- **Fault Tolerance:** Moderate. The microservice introduces a component that needs to be highly available. If it fails, pods might not get updates (unless they fall back to direct polling). Reliability can be improved by running multiple instances (with a leader election or shared cache) so that if one instance dies, others continue serving. Because the service is stateless aside from its cache, it can be restarted without losing critical information (or use an external cache to persist state). In case of Grid Engine failures or network issues, the service can implement retries and backoff when polling the external system without affecting the pods. This decoupling means pods are insulated from transient errors – they simply get the last known status from cache or an error if the service is down. According to distributed caching principles, a pull model like this is resilient to failures (neither the pods nor the cache service maintain long-lived state per client) www-users.cselabs.umn.edu. Still, the single point of failure risk means we should treat the proxy as a critical service (add liveness probes, consider a fallback mechanism). Overall reliability can be very high with proper redundancy and error handling in the proxy.

- **Latency and Responsiveness:** Slight trade-off. The timeliness of updates depends on the proxy's polling frequency. If the proxy polls the Grid Engine, say, every 5 seconds, pods might see up to ~5s delay in status changes. This is slightly slower than an instantaneous push, but typically faster than each pod polling on longer intervals. In practice, the proxy can afford to poll relatively frequently since it's the only one querying the Grid Engine – frequent polls from one source are often better than frequent polls from many clients. If the Grid Engine has a way to subscribe to events or get immediate notifications, the proxy could even adopt a push or watch mechanism internally to improve latency. Without that, this approach still provides eventually consistent updates with low latency for most use cases. The cache ensures pods get immediate responses to their queries (no waiting for an expensive external call each time). In summary, job status changes propagate

to pods as fast as the proxy updates its cache. You'd tune the polling interval to balance load vs. freshness. (For example, Kubernetes recommends using watch or at least narrow queries instead of broad frequent polls to keep latency low without overloading the server docs.aws.amazon.com/docs.aws.amazon.com.)

- Feasibility (Standard Libraries Only): High. This approach is very feasible under the constraints. Pods just need the ability to make HTTP requests (which can be done with Python's built-in libraries). No special software or complex client is required in the pods. The heavy lifting (interfacing with Grid Engine, caching) is all in the microservice, where you have more freedom to use needed libraries or even a different language. This separation aligns well with Kubernetes best practices – offloading complex integration logic to a dedicated service (or operator) rather than bloating the individual pods. Given that no new software runs inside the pods, this approach meets the requirement and is relatively easy to adopt incrementally (you can introduce the proxy without major changes to the Grid Engine or the pods' base images).

Real-World Notes: This approach is analogous to how Kubernetes controllers work: they watch the cluster state and maintain a cache, so clients querying the controller get fast, consolidated information instead of each client hitting the database. In fact, experts suggest inserting a caching proxy for heavy API queries – “better off with a proxy cache that either watches or polls... [depending] on your application”

stackoverflow.com

. A similar concept could be applied here. A concrete example is the “Bridge Operator” for HPC: a Kubernetes operator was built to submit jobs to an external HPC scheduler and act as a proxy for monitoring those jobs

arxiv.org

. In one implementation, a controller pod submits the job and then enters a loop polling the external scheduler for status, updating a Kubernetes ConfigMap with the job state

arxiv.org

. This is effectively a microservice performing polling on behalf of the cluster, very much in line with Approach 1 (the difference being it uses a Kubernetes custom resource to mirror external jobs). The use of a shared cache or state (like the ConfigMap in that example) means other components can quickly get job info without hitting the HPC scheduler directly.

Design Pattern: The caching/proxy approach follows a Cache-Aside (or Cache-Proxy) pattern. All pods fetch data from a single source which keeps data locally fresh by periodically pulling from the Grid Engine. This provides consistent reads for all pods

stackoverflow.com

. You could further enhance it by using a distributed cache (e.g. Redis) if you have multiple instances. Also, this approach aligns with the “ambassador” or “adapter” pattern in microservices – a dedicated service that adapts an external system to your cluster. It's a straightforward solution that embraces Kubernetes' principle of small, single-responsibility services.

2. Event-Driven Push Notifications

An event-driven approach eliminates polling by having the Grid Engine (or an intermediary) push job status updates to consumers. For example, when a job finishes or its status changes, an event is published to a message queue, or a webhook sends data to the interested pods/services. Pods would receive notifications in real-time, via callbacks, WebSockets, Server-Sent Events (SSE), or another messaging system.

- **Implementation Difficulty: High.** This is the most complex option. It requires an event distribution infrastructure. If the Grid Engine itself can't natively push events, you'll need to build a component that watches the Grid Engine (or its logs) and emits events. This could be a daemon that subscribes to scheduler notifications or periodically polls internally and then broadcasts changes (essentially combining a poller with push delivery). On the Kubernetes side, pods must be able to receive or retrieve those events. This might involve setting up a message broker (Kafka, RabbitMQ, NATS, etc.) or using Kubernetes-native events/CRDs. For example, one could implement a publisher/subscriber model: a central service posts job status messages to a channel, and pods subscribe to that channel. Technologies like Apache Kafka are well-suited – they're distributed and reliable, aligning with cloud-native principles stackoverflow.com– but integrating them means extra moving parts and clients. Pods written in Python would need a client library or to use WebSocket/SSE connections (the Python standard library alone has limited support for these, so you might have to include a lightweight library or use raw sockets). Overall, the development effort is significant: you must design event schemas, ensure delivery reliability, possibly handle message persistence or retries, and modify pods to handle asynchronous updates (which might be a different programming model than simple polling loops). This approach also requires more devops work to deploy and maintain the event infrastructure (brokers, or an extended operator). It's powerful, but certainly the hardest to implement from scratch.

- **Scalability: Excellent.** When done right, event-driven systems scale very well for large workloads. Instead of every pod frequently querying, you have a constant-time push per update. The load on the Grid Engine is minimal – it only sends an update when something changes, not responding to continuous polls. Likewise, the load on the Kubernetes side can be kept low: for example, using a single topic that all pods listen to, or using Kubernetes API server events that pods (or a controller) watch. Modern event brokers (Kafka, etc.) handle tens of thousands of events and consumers easily. A push model also avoids the “herd effect” of synchronized polling. In fact, Kubernetes itself prefers event watches over polling for efficiency – “watching for changes... instead of polling... further reduce[s] the load by using a shared cache” docs.aws.amazon.com. With push notifications, as the number of pods grows, the system sends out more events, but it's still roughly proportional to the number of actual state changes, not the number of pods. This can dramatically reduce redundant work. A study comparing push vs. poll for real-time updates found that a push/SSE approach not only achieved lower latency but also supported more than double the number of clients than polling in a scenario with 25k clients requiring instant updates brechtvdv.github.io. That indicates how well push can scale when immediate responsiveness is needed. However, one must consider the overhead of maintaining many open connections (if using WebSockets/SSE). Each pod might hold a long-lived connection to an event source. Efficient event brokers can handle this (Kafka through consumer groups, or an SSE server optimized for many subscribers), but a poorly designed solution (e.g. a naive webhook to each pod) could struggle. In practice, a fan-out via a broker or central service is used: the Grid Engine (or an adapter) sends one event into a channel, and that is distributed to subscribers. This decoupling ensures the Grid Engine isn't overloaded by direct notifications to N pods. Overall, for high-frequency update scenarios and large numbers of pods, event-driven communication is considered more scalable and resource-friendly than frequent polling kubernetes.io (Kubernetes' move to event-based container status updates explicitly “reduces...resource consumption” compared to polling).

- **Fault Tolerance:** Potentially high, but depends on design. A well-architected event system can be very reliable: brokers like Kafka provide durability (events are stored and can be replayed), and they handle consumer disconnects gracefully. For instance, if a pod is down or unreachable, the event might remain in a queue and be delivered when it comes back, or the system can mark it for retry. This decoupling means producers (the Grid Engine side) don't need to know if pods received the message – the broker ensures delivery or retention. That said, adding components introduces failure points: the message broker or push service itself must be highly available. If the broker goes down, events could be lost or delayed unless you have clustering or fallbacks. Another consideration is missed events: unlike polling, where a pod can always ask again, an event system must account for a lost notification (e.g. due to network hiccup). One common practice is to implement a reconciliation loop or periodic sync in addition to events – for example, Kubernetes controllers often process events but also periodically do a full resync of state as a safety net. Pods could similarly fall back to a sanity-check poll every so often (or on startup) to ensure they didn't miss anything critical. Designing idempotent, replayable events is also important (so re-processing an event or handling duplicates isn't harmful). If done carefully, this approach can be extremely reliable and exactly-once or at-least-once delivery guarantees can be achieved via the broker. In terms of system failure: if one pod fails, it doesn't affect others – each independently gets events. If the central event producer (or adapter service) fails, it's similar to the proxy service case: no new updates will flow. Mitigate this by running it in HA mode. Because the system is event-driven, there is less risk of overwhelming the Grid Engine; thus fewer failure modes due to load. In short, event-driven architectures can be made robust with techniques like persistent logs, acknowledgments, and retries stackoverflow.com, but achieving this is non-trivial. Simpler implementations (like a basic webhook) might actually be less fault-tolerant than polling (e.g. if a webhook call fails, that update is gone). Therefore, careful engineering is needed to equal or surpass the reliability of the simpler approaches.

- **Latency and Responsiveness:** Excellent (real-time). This is the main draw of push notifications – as soon as the Grid Engine knows of a status change, it can notify the interested parties. There is no need to wait for the next poll interval, meaning minimal propagation delay. In practice, the latency is on the order of milliseconds to a second, depending on network hops and broker speed. For example, if using a message queue, the pipeline might be: Grid Engine event -> push to broker -> broker fan-out -> pod receives. Each step is typically fast. Studies confirm that pushing updates yields “lower latency on the client” compared to polling brechtvdv.github.io. This is ideal for use cases where timely updates are critical (e.g. maybe you want to trigger follow-up tasks as soon as a job finishes). It's essentially real-time event-driven architecture. One caveat: if the event system batches messages or if network is unreliable, there could be slight delays, but still usually better than a fixed polling interval. Also, if an event is missed and you rely on a backup poll, that could introduce delay in those edge cases. But under normal operation, this approach is the fastest in getting job status changes to pods. In summary, notifications propagate immediately, which maximizes responsiveness. This approach shines especially when job durations are unpredictable – with polling you might guess an interval, but with events you don't have to guess; you get the update when it happens.

- **Feasibility (Standard Libraries Only):** Challenging. Given that pods can only use standard Python libraries, implementing an event client in each pod is tricky. Python's standard library supports basic networking (sockets, HTTP requests) but not higher-level protocols like AMQP or Kafka out-of-the-box. Workarounds exist: for example, pods could

use long polling with urllib to an HTTP endpoint (which is essentially a poor-man's push) or use the select socket mechanism to read from a persistent connection. But realistically, you'd want to use an existing library or client for the message system (e.g. Kafka's client, or even something like websocket-client or sseclient for SSE), which violates the "standard library only" rule. One way to keep pods lightweight is to externalize the event handling: e.g., run a sidecar container that handles the messaging and writes results to a shared volume or shared memory, which the main container (using only stdlib) can read. However, sidecars are effectively "additional software" alongside the pod, which might not be allowed either. Another approach is to use Kubernetes primitives: for instance, use a Kubernetes ConfigMap or CRD as a medium for events (the HPC event adapter updates a ConfigMap with statuses, and pods use the Kubernetes API to watch that). The Kubernetes API can be polled or watched with standard tools (you could call `kubectl get -w` in a subprocess, or use the REST API watch which is an HTTP long poll). This leverages Kubernetes as the message bus, avoiding external software. Still, it's complex to set up and not as direct as a polling loop. Because of these hurdles, the pure event-driven approach is less feasible given the constraints – it likely requires bending the "standard library" rule or introducing additional components like brokers or sidecars. In contrast, the caching service or even the hybrid polling can be done entirely with basic HTTP and timing logic. Thus, while event-driven is conceptually the most elegant, it may be impractical in the short term unless the environment allows for additional tooling. It might be something to consider for a future, more relaxed environment, or if a managed service (like a cloud Pub/Sub) can be used such that pods just use HTTPS (which is standard library compatible) to receive events (e.g. via webhooks or SSE over HTTP).

Real-World Notes: Many modern systems favor event-driven designs for decoupling and scalability. Cloud Native Patterns (Manny!) specifically advocates moving from request/response to event-driven communication to improve resilience and flexibility
stackoverflow.com

. A common implementation in industry is using Kafka or RabbitMQ – for example, a microservice architecture might publish an event when a job is done, and multiple consumers (pods) can react to it. Jonas, in a Stack Overflow answer, noted that Kafka (being distributed and cloud-ready) is a good fit for such scenarios, and suggests using a REST proxy or adapter to post events from an external resource into Kafka
stackoverflow.com

. In HPC context, true push is less common, but there are analogies: e.g., some schedulers can send emails or SNMP traps on job completion – those are event notifications that could be caught and translated into cluster events. Another example is Kubernetes itself: its architecture is fundamentally event-driven (controllers watch resources). The recent shift in Kubernetes to use CRI event-based updates for container status instead of polling is a concrete case where push proved better – it "reduces node resource consumption... compared to the legacy [polling] approach"

kubernetes.io

. This shows the efficacy of events in a high-scale system. Outside of Kubernetes, frameworks like Airflow introduced event-driven sensors (deferrable operators) to avoid constant polling of external systems, which massively improves efficiency and reduces resource usage (freeing workers while waiting for events). These real-world trends underscore that while harder to implement, event-driven push can yield significant benefits in scalability and responsiveness for large-scale or real-time workloads.

Design Pattern: The event-driven approach aligns with the Pub/Sub (Publish-Subscribe) pattern and the Observer pattern. The Grid Engine (producer) publishes events, and pods (subscribers) react. It also touches on Event Sourcing if events are stored, and Reactive design. Architecturally, you might employ a Message Broker (for decoupling and reliability) or use Webhooks/Callbacks (direct push). Kubernetes best practices suggest using watch APIs or even building a custom Operator if you need to integrate external events into the cluster. In fact, a Kubernetes Operator for external jobs could expose those jobs as custom resources and use events to update their status – under the hood it's doing event-driven updates that pods (or user code) can watch via the Kubernetes API. Patterns like “Choreography” vs “Orchestration” also come into play: an event-driven system is more of a choreographed approach where each component reacts to events rather than a central orchestrator polling for status. This can improve decoupling but requires careful design to avoid missing signals.

3. Hybrid Polling with Adaptive Backoff

The hybrid approach aims to get the best of both worlds: continue polling (ensuring simplicity and guaranteed eventual updates) but do it smartly to reduce overhead. “Adaptive backoff” means a pod will poll more frequently when updates are likely needed and back off to lower frequency when things are stable. For example, a pod might poll the Grid Engine every few seconds immediately after submitting a job (when the job might start or error quickly), then if the job is running for a long time, poll say once a minute, and perhaps increase frequency again when the job's expected end time approaches or after a certain duration. The polling interval dynamically adjusts based on some criteria (job state, time since last change, etc.). This approach can also include random jitter to avoid many pods polling in unison.

- **Implementation Difficulty: Low.** This is essentially an improvement to the existing polling logic, so it's the easiest to implement. It doesn't introduce new components; it only changes the code running in the pods. Using standard Python libraries (e.g. `time.sleep` or `sched`), one can implement exponential backoff or other adaptive timing. For instance, you might double the interval after each poll that finds no change (to a max), or maintain a schedule (poll every 5s for first minute, then every 30s, etc.). These algorithms are straightforward and well-documented – exponential backoff in networking is a common technique to reduce load and contention [en.wikipedia.org/en.wikipedia.org](https://en.wikipedia.org/wiki/Exponential_backoff). In our scenario, the logic might be: “keep polling while job is in queue or just started (fast changes possible), once job is running, back off polling to a lower rate, but if the job is near completion or if user really needs the result, maybe poll more often or allow a manual refresh.” This can all be done inside the pod's Python code without additional installations. The challenge is mostly tuning the backoff strategy: deciding the min/max intervals and triggers for speeding up or slowing down. But implementing a basic exponential backoff (which increases interval up to a cap) is trivial and often just a few lines of code. Because no new infrastructure is required, this approach has the smallest devops footprint as well. It respects the constraint since only Python stdlib is used (no new software needed in pods or cluster). In short, it's a quick, pragmatic solution to alleviate polling pressure.

- **Scalability: Moderate.** Adaptive polling reduces the total number of requests made to the Grid Engine, which helps as the system scales, but it doesn't eliminate the linear growth of requests with number of pods. If you have many pods, and especially many active jobs, you will still have many polling loops hitting the Grid Engine. The improvement is that when there's little to report (jobs in long-running states), those loops slow down significantly. For example, if 100 pods each poll every 5 seconds initially but then back off to every 60 seconds, that's a 12x reduction in query rate per pod during steady state. This can

dramatically cut unnecessary traffic. A pull-based approach is demand-driven – if nothing changes, the system stays quiet www-users.cselabs.umn.edu. However, unlike the push model, in the worst-case scenario (many jobs finishing or changing state frequently), each pod will ramp up polling and the Grid Engine could still be hit by a burst of queries. So the load is proportional to number of pods (clients), though reduced by the fact that not all pods will be polling at high frequency at once. If job events are staggered, each pod's backoff schedule will naturally distribute calls over time. If events coincide, you get a spike of polls around those times. In practice, this may be acceptable if the Grid Engine can handle occasional bursts, and the overall query volume remains within limits. Compared to constant high-frequency polling, adaptive backoff greatly improves scalability: it cuts down redundant checks and lets the system breathe when things are idle. A known downside is that if you set backoff too aggressively, you risk latency (discussed below), but pure scalability (in terms of query rate) improves as you lengthen intervals. There is a theoretical point where if pods back off enough, you could support more pods than a push system that maybe keeps a constant connection per pod. For example, one study indicated that if clients can tolerate some maximum latency, a polling interface might serve more clients than push by spreading out polls brechtvdv.github.io – meaning if you slow down polling (accepting delayed updates), the server can handle more clients because it's not overwhelmed by real-time updates. This underscores that with adaptive (slower) polling, you trade latency for the ability to support more clients. In summary, hybrid polling is reasonably scalable for moderate growth and simpler scenarios, but for very large scale or highly interactive loads, it may still fall short of the efficiency of a true event-driven approach. It's a mitigation, not a perfect cure, for polling overhead.

- **Fault Tolerance: High (at pod level).** One advantage of sticking with polling (even smarter polling) is that the system remains stateless and robust to failures. Each pod independently handles its polling; there's no central coordinator that could become a single point of failure. If a pod crashes or restarts, no other component is affected (it will simply resume polling for its job status when it comes back). If the Grid Engine or network has a glitch, the pods' backoff logic can simply keep retrying at increasing intervals until the connection is restored, without any coordination needed. This approach retains the resilience of the basic request/response model – as noted in distributed system design, a pull model is “resilient to both server and replica failures” www-users.cselabs.umn.edu. There's no queue to overflow, no event backlog to lose. The worst that happens is a pod times out on a request and tries again later. Also, because pods do not rely on external infrastructure (other than the Grid Engine itself), there are fewer components that can fail. In essence, you have graceful degradation: if polling fails temporarily, the next poll will eventually succeed and update the status (with some delay). That said, one subtle aspect is ensuring the backoff does not overreact – for example, if the Grid Engine is down for a while, pods might extend their intervals a lot; when it comes back up, it could take some time before the next poll happens. But that just affects latency, not correctness. In a failure scenario, you might actually want to reset the backoff when things recover to get updates sooner. Those are tunable details. In terms of reliability of updates, polling guarantees that eventually a pod will fetch the correct status (assuming the Grid Engine is eventually consistent and retains job info). There's no risk of “missing” an update entirely – even if the first attempt fails, subsequent attempts will get it. This is a simpler reliability model than event-driven, which must store and forward messages. The trade-off is you might detect the change later than ideal, but you will detect it. Overall, the hybrid polling approach is very reliable in the sense of delivering outcomes, and has no central point of failure – each pod's

logic is isolated. The only shared risk is the Grid Engine being overloaded or unresponsive; adaptive backoff actually helps here by throttling requests when problems occur (backoff algorithms are known to help maintain some availability under heavy load by reducing request rates en.wikipedia.org).

- Latency and Responsiveness: Variable (configurable). By design, this approach will not be as immediately responsive as a push model, but it attempts to balance timeliness with efficiency. When a job is in a critical phase (e.g., just submitted or about to finish), you can configure the pods to poll more frequently, achieving low latency detection of state changes. When a job is running for hours, you probably don't need second-by-second updates, so a longer interval (with potentially minutes of delay) is acceptable. Thus, the latency is non-uniform: initially low latency, then higher latency during steady state, then possibly lower again at the end. In the worst case, if a job finished just after a pod backed off to a long interval, the pod won't notice until the next poll – which could be tens of seconds or even minutes. That is the price of reducing load. You can mitigate this by setting an upper bound on the poll interval (ensuring, say, you check at least every X seconds). Also, if there's some way to predict job duration or get a hint (maybe user provides an expected runtime), the polling schedule could incorporate that (e.g., start polling more frequently when nearing that time). Without any hints, one could implement a pattern like "slow-fast": poll slowly for most of the run, but do a quick double-check at shorter intervals periodically (just in case it finished earlier than expected). These heuristics increase complexity a bit, but are doable. Fundamentally, this approach increases average latency to dramatically reduce overhead – analogous to networking backoff which "decreases collision but increases average wait time" en.wikipedia.org. If tuned well, the latency can be kept within acceptable ranges for the application's needs. For example, if users are fine with a job status updating within, say, 30 seconds of completion, you might cap the backoff at 30 seconds. If a job finishes at time T, the longest the pod might take to realize it is $T + 30s$ (in worst case). Many HPC workflows can tolerate that, especially if jobs themselves are long-running. On the other hand, if a job fails quickly and you backed off too slowly, there could be a noticeable gap before the pod knows. So it's about compromise. Compared to constant polling, the initial responsiveness can be identical (since you don't back off immediately), so short jobs or early failures are caught quickly. It's only during long stable periods that you accept some delay. This is generally a sound trade-off if real-time updates are not critical. It's also worth noting that if something important is happening, users or systems often trigger an earlier check manually – e.g. if a user really needs to know now, they could prompt a poll. In automated pipelines, usually a slight delay is fine. So, overall, latency is configurable with this approach: you decide the maximum delay you'll tolerate and design the backoff around that. It won't beat an event-driven push in pure responsiveness, but it can get reasonably close during key moments while avoiding unnecessary chatter the rest of the time.

- Feasibility (Standard Libraries Only): High. This approach is essentially a smarter way to do what's already being done (polling), so it fully works with just the Python standard library. No new modules are strictly required beyond what you use for polling now (HTTP client or command calls and some time calculations). It's actually the path of least resistance since it doesn't require introducing any new environment dependencies or services. Each pod's code can be adjusted and redeployed. Because of this, it's often seen as the first step to improving a polling system – you try to optimize the polling itself before resorting to more drastic architectural changes. In terms of Kubernetes integration, there's nothing special to do, which also means you're not leveraging any fancy K8s features – it's just an internal logic tweak. One possible enhancement, while staying within "standard"

means, is to use Kubernetes to help inform the backoff. For example, pods could read annotations or ConfigMap values that tell them how often to poll (which you could update cluster-wide if needed to slow down all polling during peak times). But even that isn't necessary. So feasibility is very high and the risk low: you aren't adding unfamiliar components, just improving the code.

Real-World Notes: The hybrid approach (especially exponential backoff) is ubiquitous in network protocols and client designs. TCP/IP and other network systems use exponential backoff to avoid congestion (e.g., Ethernet's collision backoff, as noted in networking textbooks, reduces collisions but at the cost of some delay)

en.wikipedia.org

. In cloud services, API clients often implement retries with exponential backoff to handle rate limits – this prevents hammering an API when it's unresponsive and gradually increases the interval between attempts. Your scenario is analogous: the Grid Engine can be seen as a rate-limited API – backoff will prevent hammering it continuously. A specific example in the Kubernetes ecosystem is how the kubelet checks container status: historically it polled the container runtime periodically, but newer versions moved to an event model because at very large scale the polling was costly

kubernetes.io

kubernetes.io

. However, for years Kubernetes did fine with periodic checks, and even now not everything is event-driven – some controllers use resync loops with backoff. The HPC Bridge operator we discussed earlier uses a fixed poll interval (configurable via a CR parameter) to monitor job status

arxiv.org

. That's essentially a manual form of this approach (though it may not adapt, it's just periodic). One could easily extend that to adaptive polling. Also, systems like Airflow sensors originally used a poke interval (polling) and later introduced a reschedule mode (which is like a sleep/backoff to free resources) – it's a similar idea of not polling constantly. While I don't have a specific public case study of "adaptive polling for job schedulers," it's a logical and common-sense method. Even users in online discussions often suggest "if you must poll, at least do it with backoff or reasonable intervals" as a best practice, to reduce load on servers

stackoverflow.com

. So this approach, while not fancy, is grounded in proven practice.

Design Pattern: This approach can be seen as an Optimized Polling or Adaptive Scheduling pattern. It doesn't have a glamorous name like the others, but it falls under techniques for efficient resource usage and Throttling. The use of exponential backoff is a known pattern for controlling retries and queries under load

en.wikipedia.org

. You might also think of it in terms of a Leaky Bucket or Token Bucket algorithm concept – effectively spacing out requests to avoid overflow. In distributed systems literature, there's the concept of "Leases" which is a hybrid push/pull strategy: a server might push updates for a lease period and then the client must poll after lease expires

www-users.cselabs.umn.edu

. Your scenario is a bit different (the server isn't pushing at all), but the idea of hybridizing approaches to balance consistency and overhead is similar. In absence of server support for leases, the client simply decides to back off when appropriate. So the design is straightforward: incorporate a feedback loop in the polling interval based on system state.

Case Studies and Examples

- **Kubernetes Controllers (Built-in Best Practice):** Kubernetes itself is instructive. Controllers use a combination of event watches and periodic resync. This means they get instant notifications but also fall back to polling every few minutes just in case. Also, Kubernetes' client libraries use informers which maintain a cache of watched resources, ensuring multiple consumers don't each poll the API server docs.aws.amazon.com. This directly mirrors our approaches: the informer with a shared cache is like a caching service plus events, and the periodic resync is like a backup poll. The result is a highly scalable system that avoids constant polling but still checks consistency periodically. Your solution could take inspiration from this: for example, implement Approach 1 (caching service) and in that service use a watch or event subscription if available, or implement Approach 2 (events) and still have pods do a rare poll as a safety net (hybrid).
- **HPC Workflow Integrations:** The Bridge Operator (research from arXiv) for connecting Kubernetes with HPC (Grid Engine, Slurm, etc.) is a real example where they chose to implement polling. They run a controller pod that submits jobs to the HPC scheduler and then enters a monitoring loop with a configurable poll interval to update job status arxiv.org. They store the status in a Kubernetes object (ConfigMap), which could then be read by others. This is essentially Approach 1 (a proxy that polls) combined with using the Kubernetes API to expose the data. The reason they likely chose polling is simplicity and reliability. This shows that in practice, a caching/polling service is a viable approach and has been used in HPC-K8s integrations.
- **Apache Airflow Sensors:** Initially, Airflow used sensors that polled for external conditions (like a file arrival or job completion), possibly with a time interval and optional exponential backoff. This was easy to implement but could tie up worker resources. In recent versions, Airflow introduced Deferrable Sensors/Triggers – essentially an event-driven mechanism where the sensor doesn't hog a worker and instead defers until an event is received (or a periodic check triggers). This migration from polling to an event loop improved Airflow's scalability for long-wait tasks. It parallels the trade-off here: moving to events improved efficiency but required changes in architecture. However, Airflow still provides both options (polling sensors with backoff for simpler setups, and event-driven triggers for high scale or when efficiency is paramount) astronomer.io. This dual approach suggests that it's valuable to implement adaptive polling first (for simplicity), and have an eye toward event-driven if bottlenecks arise.
- **Web Applications (Comet, SSE vs Polling):** In web development, a classic scenario is updating clients with new data. Techniques evolved from periodic AJAX polling to Comet (long polling) to Server-Sent Events and WebSockets. Early on, many sites just polled the server every few seconds (simple, but heavy). Over time, long-lived connections became the norm for real-time apps (chat, notifications) due to lower latency and server load. However, even today, some applications use hybrid approaches: e.g., a client might poll every 10 seconds as a fallback in case a WebSocket disconnects. The lessons from web tech echo here: use push for real-time needs, but maybe keep a backup poll. Also, caches like HTTP reverse proxies and CDNs are used to offload repeated polling of the same data – analogous to our caching microservice. So, these patterns are time-tested in other domains.
- **Academic Findings:** The article on live open data interfaces quantitatively comparing push vs pull found SSE (push) to be superior in low-latency scenarios, but interestingly noted there is a threshold of tolerated latency where a polling strategy can

serve more clients cost-effectively `brechtvdv.github.io` `brechtvdv.github.io`. This kind of analysis might guide you: if your users/job workflow can tolerate, say, 30s latency, you might stick with polling/backoff and handle a large number of pods easily by tuning the interval. If you need sub-second latency on status changes, an event approach is the way to go. In any case, that study also observed that push had lower CPU usage on the server and supported more clients for real-time needs, reinforcing that event-driven is generally more efficient if your system and implementation can handle it.

Suggested Architecture and Design Patterns

Depending on which approach (or combination) you choose, the architecture will vary:

- For Caching/Proxy Service: A suggested architecture is a central Job Status Service within your cluster. It could be deployed as a Deployment with, say, 2 replicas behind a Service. Pods that need job info make a request to this Service (e.g., HTTP REST GET `/jobs/<id>/status`). The Job Status Service maintains a cache (in-memory or backed by Redis for durability). It regularly polls the Grid Engine (perhaps each job or queue in a round-robin fashion) and updates the cache. Optionally, it could expose a watch/subscribe endpoint (WebSocket or SSE) for Approach 2 in the future, allowing a smooth transition to events without changing the data source. This service should be stateless (if using Redis or similar) or use leader election (if only one should poll to avoid duplicate load). This pattern is akin to an API Gateway or BFF (Backend-For-Frontend) but specialized for polling an external system. It adheres to the Single Responsibility Principle: one service handles interaction with the Grid Engine. This also follows the Facade pattern – it presents a simplified interface to pods, hiding the complexity of the Grid Engine. In Kubernetes terms, ensure this service has proper resource requests (it may need some memory for caching and CPU for polling threads), and use liveness/readiness probes to keep it healthy. Also, secure it (if needed) so that only authorized pods/namespaces can query it, since it might be exposing job info cluster-wide.
- For Event-Driven Push: A possible design is an Event Broker + Notification Service. The Notification Service would connect to the Grid Engine or monitor its output (similar to the cache service, but instead of serving queries, it pushes events). For instance, when it detects a job state change, it publishes an event to a broker topic (like “jobUpdates”). The pods could either connect directly to the broker (if they can have a client) or you deploy a lightweight Push Gateway in the cluster that pods can connect to via WebSocket. The Push Gateway would subscribe to the broker and then forward messages to connected pods. (Think of it like a WebSocket server that the pods connect to; when it receives a job update message, it broadcasts to the relevant pod or all pods if filtering by job ID is needed.) If using Kubernetes events, the Notification Service could create/update a Custom Resource (e.g., JobStatus CRD) for each job and set its status; then pods (or their sidecars) could watch those using the Kubernetes API. The architecture here is more involved: you’d need to run and maintain the broker or rely on an external one. Knative Eventing or Argo Events are CNCF projects that could help implement this pattern on Kubernetes – they provide scaffolding for event brokers and triggers. Design patterns at play include Observer(pods observe the event source), Publisher-Subscriber, and possibly Event Sourcing if you log all events. A critical part of this architecture is making sure events are delivered exactly once or at least once. Using acknowledgments from pods back to the broker (or some stable storage) can help track that. Another consideration is how pods know which events pertain to them – likely by job ID. You might design the events to carry a job identifier, and the pods only react to the ones for their job (filtering can be done client-side or via separate topics/rooms per job). Kubernetes best practice: try to leverage existing platforms (don’t

reinvent a whole queue system if you can avoid it). If your organization already has a messaging system, integrate with that. Otherwise, starting with a simpler push mechanism (like long polling or SSE from the proxy service) might be an intermediate step.

- **For Hybrid/Backoff Polling:** The architecture here doesn't introduce new components, but you should enforce some coordination to avoid synchronization issues. One pattern is to incorporate jitter: when backing off, add a small random delta to the sleep time so that pods don't all wake up at the same exact intervals (which could cause periodic bursts). This is a known practice to avoid thundering herd problems. Another architectural consideration is monitoring and adjusting the backoff parameters globally. You might implement a way to tune backoff without rebuilding the image – for example, pods could read configuration from a ConfigMap (poll intervals, backoff factor, max interval). This way, if you find the cluster is still overloaded, you can increase the intervals by updating the ConfigMap, and pods will pick it up (if they check it periodically or if you restart them). This adds a touch of dynamic control. Design-wise, this approach is simply an enhancement of the Polling Loop pattern. You might document it as a state machine: On job submission -> state = "pending", poll fast; state = "running", poll slow; state = "finishing" (maybe if job runtime > threshold or check more frequently near end) -> poll fast again. Implementing this state-based polling in code with clear transitions will make it easier to maintain. It's also wise to instrument this: log the polling interval changes and perhaps count how many polls each pod does, so you can verify reduction in load. From an ops perspective, ensure the Grid Engine can handle at least the worst-case polling frequency times number of pods, or adjust accordingly. If there's a risk of even backoff polling overwhelming it, you might need to impose a hard rate limit (like each pod will never poll faster than X, and also if too many jobs, perhaps they could coordinate – though coordination between pods complicates things and usually isn't done). Typically, the Grid Engine is quite capable of handling status queries, but it depends on its internals.

Kubernetes Best Practices and Further Recommendations

- **Leverage Kubernetes APIs and Operators:** If possible, consider encapsulating the Grid Engine integration in a Kubernetes Operator. This would effectively turn external jobs into custom Kubernetes resources. The operator can implement Approach 1 or 2 internally (e.g., poll and update CR status, or push events to CR status updates). Kubernetes clients could then simply watch these CRs. This is in line with K8s best practices of extending the platform using CRDs for custom automation. It keeps your solution cloud-native and might reduce the custom wiring needed in pods. The operator approach was demonstrated in research arxiv.org and could be a long-term solution.

- **Gradual Evolution:** You don't have to pick one approach for all time. You could start with Hybrid Polling (immediate relief to Grid Engine load with minimal effort). Then introduce a Caching Service to further centralize and reduce load – pods can switch to using the service one by one. Finally, you could evolve that service to support push notifications (for example, add WebSocket support or use it as an event source to feed into a message broker). This evolutionary path follows the principle of continuous improvement. Each step provides value and sets up the next: backoff polling fixes the urgent inefficiency, the cache service provides a platform for more optimization, and that platform can then be enhanced with events when ready.

- **Monitoring and Metrics:** Whichever approach you choose, implement monitoring. Track the number of Grid Engine API calls, the latency of those calls, and the cache hit rate (for Approach 1). If using events, monitor event queue sizes and delivery times. In a Kubernetes environment, tools like Prometheus can scrape metrics from your

proxy service or even from pods (you could increment a counter each time a pod polls). This data will help validate improvements – e.g., you should see a drop in polling calls after implementing backoff or a caching layer. Also watch the load on the Grid Engine node – CPU usage, etc., to ensure it's improving.

- **Testing Scalability:** Before fully rolling out an event system, test it in a staging environment with a simulation of many pods/jobs. Event architectures can have quirks (like tuning Kafka partitions or WebSocket timeouts). Ensure the solution holds up with high concurrency. Similarly, test the caching service under load – if 1000 pods ask for status at once, does it serve from cache efficiently? Does it avoid stampeding the Grid Engine with simultaneous refreshes? These tests will inform the poll frequency or need for further tweaks (like maybe a small delay when multiple requests for the same job come in concurrently, etc.).

- **Consider Data Freshness vs Load:** For each approach, decide what the acceptable staleness of data is. With events, ideally none (real-time). With caching, maybe a few seconds. With polling/backoff, maybe up to a minute. Communicate this to stakeholders. It might turn out that everyone is fine with 30-second-old status info if it means the system is stable. Or maybe you discover certain jobs (like very short ones) need a different treatment (perhaps those always use a fast poll or a different channel). Possibly adopt a tiered approach: e.g., short jobs (expected <2 min) use frequent polling (or a different mechanism) because they benefit from it, whereas long jobs use the backoff strategy. This kind of segmentation can optimize both latency and load.

- **Additional Software in Cluster (if allowed):** If the restriction on “no additional software in pods” is more about the pods running jobs (perhaps to keep them lean or not modify user environments), deploying additional components in the cluster might still be acceptable (since those are not user job pods). If so, using something like Redis for caching or RabbitMQ for messaging could be very helpful. They are well-tested tools for these problems. For example, Redis could be used by the proxy service to store job statuses with an expiration; multiple instances of the service could share that. Or RabbitMQ could be used to broadcast job events, which a lightweight Python script (even using standard pika or HTTP API) could consume. Weigh the operational overhead: running these services is an added burden, but if your environment has them or can manage them, they can offload a lot of custom logic.

- **Security and Access:** If pods currently connect directly to the Grid Engine (maybe via its API or CLI with credentials), moving to a centralized approach means handling credentials there. Ensure that the proxy service or event producer has the appropriate credentials and that you're not exposing sensitive data. Kubernetes Secrets can be used to store any auth tokens or passwords needed to query the Grid Engine, and mount them to the service. If using events, consider whether any sensitive info is being broadcast (and if so, use secure channels or filtering per consumer).

- **Further Research:** Look into projects or tools that might already solve parts of this problem:

- **Liqo or other multi-cluster solutions** – sometimes they address offloading workloads (though Liqo is more about offloading pods to other clusters, not exactly this, but worth a look) docs.liqo.io.

- **Batch Job Operators** – there might be existing open-source operators for systems like SGE or Slurm that you could adapt.

- **Kubernetes Event-driven Autoscaling (KEDA)** – while KEDA is about scaling on events, it shows patterns of hooking event sources to Kubernetes; for example, KEDA

can listen to a queue and then take action, which is somewhat analogous to what you'd do for events from Grid Engine.

- Workflow engines – e.g., Argo Workflows or Tekton, if your use case fits, maybe the entire flow could be managed at a higher level where an external job is just one step and the system watches it. These might provide their own mechanisms to poll or wait on external jobs.

- Potential Improvements: If you implement the caching service, a future improvement could be to add an in-memory publish-subscribe within it. For instance, pods could open a streaming connection to the service, and the service pushes updates over that connection when the cache changes. This avoids the complexity of a full message broker but gives you some of the latency benefits of push. It could be done with SSE (Server-Sent Events) which are essentially HTTP responses that stay open and send events. Python's `stdlib` doesn't directly support SSE, but a simple implementation could be done with Flask or a minimal `async` server. This would violate "stdlib only" for pods unless you make the pod just do a blocking read from the HTTP response (which is possible but a bit hacky). Alternatively, the service could webhook the pods (if pods run a tiny HTTP server or can expose an endpoint). That again complicates pods though. So pushing from the service to pods is tricky under constraints, which is why a central broker or direct SSE client might be better if allowed.

- Kubernetes Events for Alerting: You might also consider using Kubernetes Events (the object kind that you see with `kubectl events`). A controller could generate Events on the pod or Job object when an external job changes state. For example, post a Kubernetes Event "Job X completed" attached to some object. This could be viewed in `kubectl describe` or even picked up by a system like Argo Events or notifications (there are tools like `kubewatch` that can send events to Slack, etc. when certain events occur civo.com). This isn't a direct solution for pods to get the info, but it's a nice integration point to keep humans or higher-level systems informed without constant polling.

Conclusion & Recommendation:

Given the constraints and typical Kubernetes practices, a combination of approaches might yield the best result. In the immediate term, implementing Hybrid Polling with Backoff is recommended – it's simple and will reduce unnecessary load on the Grid Engine quickly. In parallel, you can develop a Caching Proxy Service which adheres to Kubernetes design principles (decoupling and single-responsibility). This service can be introduced gradually and can itself use efficient techniques (like watching, or at least optimized polling with its own backoff) to interface with the Grid Engine. Many teams find that a caching service alone resolves their scaling issues because it cuts out duplicate calls and lets you centralize logic (e.g., you could cache static info like cluster queue status, etc., not just individual job state). If your system requires faster updates or you want to minimize even the slight delays, then Event-Driven Push should be the eventual goal. It will provide the best scalability and user experience (real-time feedback). However, weigh the complexity: it's probably justified if you foresee a large scale (hundreds of jobs, many users) or if users are demanding instantaneous updates. If you do go that route, consider using established patterns like a message queue or the Kubernetes API itself to convey events, to avoid reinventing too much from scratch.

To summarize the comparison in a few key points:

- Easiest to Implement: Hybrid Polling (Adaptive Backoff). Minimal changes, no new components – just adjust the polling logic in pods.

- **Most Scalable & Responsive: Event-Driven Push.** Especially for real-time needs and heavy load, push beats polling in latency and reduces overall system strain [brechtvdv.github.io](https://github.com/brechtvdv/iokubernetes.io), at the cost of more complex infrastructure.
- **Balanced Middle Ground: Caching/Proxy Service.** Moderate effort but aligns with Kubernetes patterns; improves scalability by caching and consolidating requests stackoverflow.com, and can be made reasonably fault-tolerant. Latency is slightly higher than pure push but often acceptable.
- **Fault Tolerance:** Polling (and thus hybrid) is inherently stateless and tolerant to failures www-users.cs.umn.edu; the caching service adds a manageable point of failure (mitigated by HA deployments), whereas event systems add new failure modes but can be built to be reliable (with durable queues, etc.).
- **Feasibility:** Both the caching service and hybrid polling are very feasible with standard tools. The event approach, while conceptually ideal, may require relaxing the “standard library only” rule or introducing additional services, which might not be immediately feasible.

In light of Kubernetes best practices, you should aim to minimize tight polling loops (they are an anti-pattern at scale)

docs.aws.amazon.com

. Using watches/events or at least a shared polling mechanism is the recommended path.

Thus, moving towards either Approach 1 or 2 is advisable for the long run. A potential roadmap is: implement Approach 3 now, design and deploy Approach 1 next (with perhaps an informer-like mechanism), and investigate Approach 2 for the future if needed.

Further Research: Explore the Kubernetes documentation and community discussions on scaling and event handling – for instance, the AWS EKS Best Practices guide emphasizes using shared informers (watch + cache) over polling for custom controllers

docs.aws.amazon.com

, which reinforces the direction you’re taking. Look into CNCF projects (like Argo, Knative) for event-driven examples. And if possible, reach out on forums (Stack Overflow, Kubernetes Slack) with your specific use case – others running HPC workloads on Kubernetes may have insights or even open-source solutions. By iterating carefully and leveraging proven patterns, you can incrementally achieve a robust, scalable system for monitoring Grid Engine jobs without burdening your pods or the scheduler.