BASICS OF SYSTEM DESIGN

↳ Piyush Garg - II.

Video summary.

(00:00–00:33) Introduction to Part 2: The video aims to explore deeper into system design — focusing on scalability, containerization (e.g. containers, serverless), and newer architectural components like Kubernetes and containers.

(01:03) Recap of fundamentals: horizontal vs vertical scaling; microservices, API gateways, load balancers, queue-based asynchronous processing, fan-out architectu and queue systems.

(02:05–02:37) Key insight: System design depends heavily on a company's unique use case, traffic patterns, and goals. There's no "one size fits all." The balance is between building a fault-tolerant, scalable system and optimizing cost — blindly adding resources isn't ideal.

(03:09–04:10) Example discussion: Even for systems that look similar on the surface (e.g. video streaming platforms like YouTube, Netflix, Hotstar), their system desig vary dramatically because their traffic patterns & usage behaviours differ. You can't just copy one design and expect it to fit another.

(05:44–07:19) Challenge of sudden spikes: If traffic increases gradually, horizontal scaling (adding more servers over time) works. But sudden spikes ("burst traffic") ca break this approach — because average CPU/memory-based autoscaling policies react too slowly, risking overload.

(08:22–10:28) For predictable events (e.g. movie release, sale day, scheduled content launch) platforms can pre-scale or "warm up" servers ahead of time — for instar scaling up many servers hours before a known spike to handle the load, then scaling down when spike passes.

(11:04–12:39) For unpredictable traffic (e.g. live events, viral content) — where demand can spike arbitrarily — traditional scaling strategies fail. You need advanced mechanisms to handle unpredictable load surges.

(13:14–16:47) Mixed-service systems (e.g. VOD + live streaming) need differentiated scaling strategies: for example, scaling live-streaming services heavily when a liv event starts, while scaling down other services to optimize resource usage.

(17:16–18:57) Even user behaviour patterns can cause hidden load spikes: e.g. many users returning to a "home" or "catalog" screen simultaneously — causing API lo — so you must anticipate not just streaming load but other API requests too.

(Conclusion ~19:29) Takeaway: Designing scalable, robust systems isn't about copying architecture diagrams — it's about continually monitoring, learning patterns, understanding traffic, and optimizing over time. It evolves with real usage.

⚙️ Context & Relevance (with external insights)

The video's differentiation between horizontal vs vertical scaling, and preference for horizontal scaling for large-scale systems matches the general definition of scalab Horizontal scaling (adding servers) and vertical scaling (adding resources to one server) are classical approaches in software architecture.
DEV Community
+1

The use of container orchestration — notably Kubernetes — for scalable, resilient containerized deployments is well established. Kubernetes enables horizontal autoscaling (pods or nodes can auto-scale based on CPU/memory or custom metrics), self-healing, load balancing, and efficient management of containers across clusters.
DataCamp
+2
Wikipedia
+2

The trade-off between performance, scalability, and cost — as emphasized in the video — reflects broader best practices: a system must be designed not only to hand peak load, but also to remain cost-efficient and fault tolerant over time.
testRigor
+1

If you like — I can extend this summary into a cheat-sheet format (with key rules, do's & don'ts, when to use containers/Kubernetes, scaling policy templates) that you use for system-design prep (bearing in mind your background in backend/web dev).
Learn more on Glasp: https://glasp.co/reader?url=https://www.youtube.com/watch?v=YuB3OuF3MUE