MANJIT SINGH

Toronto, ON M5J0B5 | +1 (514)-549-1485 | manjitsingh07.1998@gmail.com | linkedin.com/in/manjit-singh-705996164

Professional Summary

- **Senior Software Engineer (Java 17 / Spring Boot 3)** with ~3 years of experience building **high-availability, mission-critical back-end systems** for capital markets and financial services.
- **Expert in microservices architecture, REST APIs, and event-driven messaging** (**Solace PubSub+**, MQBridge), with deep experience in **CI/CD automation** using GitHub Actions & JFrog Artifactory.
- **Proven record of cutting market data latency, saving six-figure annual vendor costs, and driving platform reliability** through robust observability (**ITRS Geneos, Grafana**).
- Strong background in **containerization** (**Docker, Podman**) and **Linux automation** (**AutoSys, Shell**), delivering seamless deployments and operational efficiency.

Skills

- **Languages / Tools:**Java 17, Python, Shell, Maven, JUnit 4/5, Git, IntelliJ IDEA
- **Frameworks / Infra / Platforms:**Spring Boot 2/3, Spring MVC, Spring JDBC, REST APIs, Catalys FIX Engine, Apache Tomcat, Docker, Podman, Design Patterns, Unix/Linux
- **Messaging / Integration / Data:**Solace PubSub+, Solace MQBridge, FIX, SFTP, Kafka, Oracle DB, Databricks, Redis, NGINX, F5 Load Balancer
- **DevOps / Monitoring / Scheduling:**GitHub Actions, JFrog Artifactory, AWS, Confluence, AutoSys, ITRS, Grafana

Experience

Senior Software Consultant – Fixed Income, Capital Markets **09/2023 to 06/2025**
CIBC **Toronto, Canada**

- **Modernized and re-architected a suite of event-driven microservices (Java 8 → 17, Spring Boot 2 → 3),** reducing average API and job execution times by **20%**, containerizing applications with Docker/Podman for rapid scaling, and improving uptime and cost-efficiency by migrating from IBM Solaris to RHEL servers.
- **Enhanced system scalability, security, and reliability** by introducing NGINX and F5 load balancers for high availability, implementing IP whitelisting and audit logging for secure API access, and supporting maintainability through thorough system documentation.
- **Migrated 300+ AutoSys jobs & shell scripts during the IBM Solaris → RHEL cut-over,** ensuring zero downtime and boosting production workflow availability to **99.99%** for critical batch and integration processes.
- **Reduced market data latency by 30%** through end-to-end event-driven processing: integrated Bloomberg FIX streams (Catalyst FIX → Solace Topics), and implemented Redis caching to optimize bond price distribution—accelerating thousands of **price updates from 6 minutes to 4 minutes** and enabling real-time data propagation.
- **Designed and implemented data streaming architectures** for Settlement, Allocation, and Inter-company Messaging using Solace PubSub+, secured external data flows via Solace MQBridge and SFTP.
- **Decommissioned legacy OMGEO (Allocation Trading System) and GLOSS & ARROW (Back-Office Settlement Systems);** introduced modern SFTP + ARROW-based microservices, saving **over $100K in annual costs** and modernizing data exchange patterns.
- **Co-authored a pluggable, self-service reporting microservice framework**—enabling BAs to onboard and schedule custom data extracts (CSV, XML, JSON) for external consumption via SFTP, email, or API, without dev intervention.
- **Designed, developed, and maintained RESTful APIs consumed by 10+ internal teams, processing over 10,000+ daily transactions across trades, positions, securities, and prices with 99.99% uptime,** enabling reliable access to critical capital markets data and powering automated integration workflows.
- **Developed comprehensive JUnit 4/5 test suites (90%+ coverage)** for all core microservices, integrated into CI/CD (GitHub Actions, JFrog Artifactory) for automated, production-grade releases.
- Leveraged **GitHub Copilot** and **CIBC's proprietary LLM-based AI tools** to accelerate development workflows, automate repetitive coding tasks, and enhance code quality—resulting in faster feature delivery and improved team productivity.

Functional QA / Technical Tester – Virtual Reality (VR) **03/2023 to 09/2023**
Keyword Studios **Montreal, Canada**

- Designed & executed test plans for **Meta Quest (Oculus) VR titles** in Unity/C#, using Quest dev tools for performance telemetry.
- Logged reproducible defects with **"Action – Expected – Result"** Jira titles; prioritized severity and tracked **100 +** issues across sprints.
- Built comprehensive regression suites and collaborated with engineers to validate hot-fixes in CI builds.
- Facilitated cross-disciplinary daily syncs (design, QA, engineering), accelerating **bug resolution** and ensuring on-time milestone delivery.

Software Developer & Machine-Learning Intern **01/2020 to 06/2020**
SASE Laboratory, DRDO **Chandigarh, India**

- Developed a Python-based backend service and automated ML pipeline to predict **snow-avalanche risk** for Indian Army bases using KNN, SVM, and ANN (**83 % accuracy**).
- Implemented **data ingestion and auto-preprocessing flows** from high-altitude sensors, enabling real-time decision support.
- Built a GUI-based model configuration interface allowing users to select algorithms and train models using **(5/10/20) years** of historical data.
- Designed and **deployed scheduled pipelines** to generate daily avalanche forecasts and **CSV** reports.
- Delivered **visual dashboards** in Jupyter using Matplotlib for defense analysts to interpret risk scores.

Education

MEngg.: **Software Engineering** **08/2022**
Concordia University Montreal

B.E.: **Computer Science & Engineering** **09/2020**
Punjab University Chandigarh

Manjeet Singh – Complete Experience Portfolio (Detailed Narratives + Refined Points)

Software Developer & Machine Learning Intern – Snow Avalanche Study Establishment (SASE), DRDO, Chandigarh, India (Jan 2020 – Jun 2020) I worked at the Snow Avalanche Laboratory (DRDO) on a defense-critical project to predict snow avalanches in the Himalayas for Indian Army planning. Full Story: • Instruments installed in high-altitude avalanche-prone regions recorded weather parameters (temperature, humidity, wind speed, snow depth) and transmitted them to Chandigarh. • Built automated data pipelines for cleaning and preprocessing 30+ years of sensor data. • Faced highly imbalanced datasets (1:1000 avalanche to non-avalanche). Using Python's imblearn package and resampling techniques (e.g., SMOTE), I balanced it to 1:10, creating synthetic samples and drastically improving accuracy. • Developed ML models including ANN, SVM, KNN, Logistic Regression, and Linear Regression, reaching 83% accuracy. • Built a GUI where scientists could select a date and receive avalanche probability forecasts. • Visualization & Reporting: Designed dashboards in Jupyter (Matplotlib/Seaborn), scheduled daily forecasts, and exported CSV reports. • Collaboration: Worked with high-level scientists, engineers, and soldiers to ensure solutions were practical and usable. • Impact: Enabled the Army to plan troop movements in avalanche-prone zones more strategically, saving lives and improving logistics.

Functional QA / Video Game Tester – Keyword Studios, Montreal, Canada (Mar 2023 – Sep 2023) At Keyword Studios, I worked as a tester focusing on Meta Oculus VR titles developed in Unity/C#. Full Story: • Tested VR games extensively, checking gameplay, performance, and usability issues. • Collaborated with developers to reproduce and validate fixes for critical issues. • Reported bugs in Jira in structured format: Action (steps), Expected Result, Actual Result. • Contributed to regression testing cycles, sprints, and daily standups in Agile environment. • Specialized in VR testing (motion tracking, immersion quality, device stability). • Impact: Delivered stable and immersive VR experiences by ensuring critical issues were resolved before release.

Senior Software Consultant – Fixed Income, Capital Markets, CIBC, Toronto, Canada (Sep 2023 – Jun 2025) I modernized mission-critical systems in CIBC's Fixed Income Capital Markets division, working across migrations, market data optimization, trade settlement, reporting, and security. Full Story: • Modernization & Migration: - Migrated multiple apps from Java 8 → Java 17 and Spring Boot 2 → Spring Boot 3. - Migrated infrastructure from IBM Solaris servers to RHEL Linux servers. - Built Docker images and deployed apps on Podman rootless containers, cutting boot times and job execution by ~20%. - Improved security with F5 load balancers, reverse proxies, and IP masking. • Batch & Job Orchestration: - Migrated 300+ AutoSys jobs from Solaris to Linux. - Designed FileWatcher jobs across two servers (Panda & Second Gate), triggering downstream jobs only when files arrived on both servers. This enabled load branching and removed single-server dependencies. • Market Data Latency Optimization: - Initial system polled the DB every minute for Bloomberg updates before sending to FISBOND vendor. - Replaced DB polling with Redis cache lookups, enabling instant comparisons and faster propagation. - Next, bypassed DB entirely by directly connecting Bloomberg streams to the FISBOND Solace pipeline, achieving near real-time updates. - Reduced vendor data latency from 6 minutes to under 4 minutes. • Trade Settlement & Allocation: - Decommissioned Broadridge GLOSS (costing >$100K annually) and migrated to Paramax Arrow microservices for trade settlement. - Developed microservices to route trades by region and business type for settlement. - Decommissioned Omgeo allocation system, migrated

allocations to SFTP-based transfers, simplifying architecture and reducing vendor reliance. • Reporting Automation – Pluggable Framework: - Business analysts and finance/audit teams relied on devs for custom reports. - Built a self-service reporting framework with GUI configuration: report name, data source (SQL, API, file), output format (CSV, XML, JSON), and delivery (Email, SFTP, FeedHub). - Analysts could create reports “on the go,” automating 100+ reports without dev intervention. • Software Engineering Practices: - Followed SOLID principles and ACID properties. - Implemented Strategy, Observer, Singleton, and Chain of Responsibility patterns to ensure decoupled, maintainable, and scalable code. • Collaboration, AI Productivity & QA: - Partnered with BAs, QA, and support teams to translate business requirements into robust implementations. - Used GitHub Copilot and proprietary LLM tools to improve productivity by ~30%. - Achieved 90%+ test coverage using JUnit 4/5. • Security & High Availability: - Configured F5 load balancers and NGINX reverse proxies for availability and failover. - Introduced IP whitelisting to secure production APIs. - Implemented Blue-Green deployments for zero-downtime rollouts. • Event-Driven Architecture & Messaging: - Designed event-driven integrations using Solace PubSub+ and MQBridge. - Ensured secure vendor communication via MQBridge proxy, isolating internal Solace topics from external vendors. - Enabled scalable real-time trade, price, and position distribution. • CI/CD Automation: - Integrated services with GitHub Actions and JFrog Artifactory pipelines. - Automated build, test, snapshot/release deployments with version traceability. - Reduced manual steps and accelerated delivery cycles. • Crisis Management: - Handled a 2 a.m. production outage caused by Bloomberg file misnaming in date-roll mechanism. - Diagnosed issue, coordinated with QA for rerun, restored job chain. - Implemented FileWatcher safeguard to prevent recurrence. • Results & Impact: - 99.99% uptime for production services. - Reduced data latency, cut vendor costs, automated reporting, improved security posture. - Built scalable, maintainable systems supporting capital markets trade lifecycle end-to-end.

Let's start with the DRDO experience. I was working at the Snow Avalanche Laboratory in Chandigarh. And what was happening, there are scientific tools that were installed on the high-altitude snow avalanche-bound regions. So those tools used to record multiple weather parameters. And then those parameters are reported back to the laboratory. And then based on those parameters, we obtained the data, clean the data, build data, cleaning pipelines, optimize the data. And then we built multiple machine learning models, such as using artificial neural networks, support vector machine, k-nearest neighbors, logistic regression, linear regression, all sort of algorithms. And then we created a GUI, which a scientist can go on any particular date, select what will be the snow avalanche. And it will say what is the percentage of a snow avalanche. And we achieved 83% accuracy. So we built those models to predict whether a snow avalanche will occur or not, so that the Indian Army can strategically decide where to go based on the readings from this laboratory.
Okay, so adding more to it. So, the data was highly imbalanced because Snow Blanche is very rare and we were recording it for like last 30 years. So, like the imbalance ratio is 1 to 1000, 1 is to 1000. So, we used imblearn package in Python to balance the data to create the artificial points. We use different imbalance learning algorithms to create the artificial data. So, the ratio is 1 is to 10 and that made the accuracy of the model really great. So, I integrated with different scientists, high-level scientists, and then the technical people and the soldiers on ground in order to build this project. So, at this point now.
So, moving on to the Keyword Studios experience. So, basically, in Keyword Studios, I was a video game tester. We used to test the video games developed in C-Sharp. We used to integrate with developers and we will play the games throughout the day and find the errors and then record them and document them and upload them in Jira and then collaborate with the developers to let them know about these issues. So, my project was specifically on the MetaOculus virtual reality game. So, we focused on the ensuring, reporting all the bugs as soon as we can, having the bug resolution, testing, and then put the Jiras in such a way that the problem is clearly defined in the three parameters, what are the actions we took in order to reproduce the error, what is the expected output we were expecting, and what is the actual output and the result that we used to show. This is what we did.
Now coming to the CIBC experience so talk about majorly and I migrated multiple applications from Java 8 to Java 17 and Spring Boot 2 to Spring Boot 3 and we were migrating from more older IBM Solari servers to Red Hat Linux servers so that's why we upgraded the technology and we also created the docker images and deployed them on podman containers on our Linux servers so this reduced the boot up time also also reduced the job execution time by 20 percent based on the QA's testing before and after and we also added extra security by having F5 load balances so our actual IPs are masked and we also use reverse proxy in Linux so that downstream teams who are accessing our APIs that load can be distributed accordingly so yeah this is one thing
Okay, here's another point. So basically, since we migrated from IBM Solaris servers to Red Hat Linux servers, the Odyssey jobs that were pointing to IBM Solaris, we need to migrate them to RHEL servers. So I migrated 300 plus jobs. And also at the same time, we introduced new architecture in which there are jobs that are dependent on files. So we create, and we had two servers, Panda and Second Gate. So we created two jobs, FileWatcher jobs, for server one and server two. So these two jobs will use to look for the same file if it arrives successfully on both

the servers. When it does, then the file-dependent jobs will run. This way, the file-dependent Odyssey job can run on any server without being dependent on any particular server and putting extra load on a particular server. So that's how we did load branching for the autosys jobs.

All right, moving on to the next point. So basically, I reduced the market data latency between the server and the vendor called FISBOND. So what was happening is the security updates used to come to us from Bloomberg, and we used to send it into the database. And now, our application, every minute, it will go to the database and see if there are any updates. If there are, it will transfer it into a format that the vendor supported, and then we used to send it over the Solace and KubeDash. Now, what was happening was database polling was happening to see if the security update has been already sent. So to compare the new value and the existing value, we used to do database polling. So to avoid that, now the update comes to us, and then we will compare it with the Redis cache, and the comparison was faster with Redis cache. So we were able to send more faster the updates. So this is one point.

Okay, now let me talk about other points. So, basically, there was another application that was receiving those security updates and then we were storing it to a database. So, what I did was for this Fisbond vendor application, I directly connected the Fisbond Solace pipeline with the Bloomberg's incoming messages so that we don't have to go to the database. Now, we are hearing that we are getting the security updates as soon as we receive them from Bloomberg. So, this way, we don't even have to connect with the database and do the database polling.

Going to another point, in Capital Markets, I was working on the allocation system and the back-office system. So, we had a back-office system called GLOSS, which was given to us by the Broadridge vendor. So, they were charging us over 100K+. So, we deconditioned that and we connected with the new vendor providing the same functionalities called Paramark, and their system name was Arrow. So, I worked on Arrow-based microservices to transfer the trades based on the region and the business type to settle them at the back-office. And also, for the allocation, we used to receive the block trade from the Bloomberg. So, we used to send it to Omzeo via the Omzeo-based application running on our server. So, we deconditioned that application and moved to SFTP Transfer Protocol. And then, we sent the file to SFTP Transfer Protocol, making Australia less reliant on Omgeo.

Back at CIBC, I built a self-serving reporting network service framework. So, due to our team heavily reliant on business analysts, different teams such as operations team, finance team, audit teams used to contact business analysts for different type of finance reports based on the trades happening every day. So, they used to create custom reports for custom teams and there were multiple reports. So, they were dependent on developers to write the query, extract the data. So, I built the pluggable platform. So, with the help of content developer, we developed a GUI on which they can mention the report name, where to store the report on the server, what are the emails of the recipients of the server, or is it SFTP transfer, what is the feed hub account that we need to send to, what is the report type, CSV, XML, or JSON. So, all the features of a report can be configured through a GUI platform. And then, is the report being, is the data coming from API? So, mention the API, is the data coming from SQL? If it's SQL, mention the SQL. If the data is coming from another file, mention the file name in the path. So, this way, every report was made, can be made on the go, and the business analysts don't need to rely on

the developers, and they can create custom reports on the go. That's why, that's what, that's the system that I made.
I also closely collaborated with business analysts, QAs, the support team, used to translate the business analyst requirement into the technical implementations. Also, I leveraged GitHub Copilot and CIBC's proprietary LLM-based AI tools, in which it made my job 30% faster. Also, we used to cover the test cases for 90% plus coverage. And, yeah, this is what I did.
I also utilize SOLID principles, ACID properties, and different design patterns such as STRATEGY pattern, OBSERVER pattern, SINGLE pattern, and CHAIN OF RESPONSIBILITY pattern to make the code highly decoupled and cohesive and follow those principles to write the clean and maintainable code.
🔍 What Is “Root” in Linux?
In Linux, root is the superuser — it has full system-level privileges, meaning it can:

Install or delete software

Access/modify any file

Start/stop system services

Change user accounts and permissions

Interact with all network interfaces, hardware, etc.

Any process running as root can do anything on the system — including potentially destructive or unsafe operations.

🐳 What Does Docker Do with Root?
Docker traditionally runs a central background process called the “Docker daemon” (dockerd).

This daemon runs as root, even if your docker run command is executed by a regular user.

So:

Your container processes may run as non-root, but Docker itself is managing them as root.

That means if there's a vulnerability inside a container, it can escape the container and potentially gain access to your host system — because Docker runs with elevated privileges.

This is called a privilege escalation risk.

🔐 What Is “Rootless Security” in Podman?
Rootless containers:
Are containers started and managed entirely by a non-root user, without needing a root-level daemon.

There is no privileged dockerd-like daemon running behind the scenes.

Each user runs their own Podman containers with only their own user’s permissions.

So:
If a rootless container gets compromised, the attacker only gains access to that non-root user’s scope (e.g., limited to a home directory or certain files), not the whole system.

🔒 Why Is This Safer?
It removes the attack vector of a central privileged daemon.

It isolates each user’s containers — there's no shared daemon, so one user’s compromise won’t affect others.

It aligns with “least privilege” security principles — users and apps should only have the permissions they truly need.

🧠 Analogy to Understand:
Imagine you're in an office building:
Docker: The building has one powerful admin (root daemon) managing everyone's rooms. If someone sneaks in using their room, they can potentially break into others.

Podman (rootless): Each person can only access and manage their own room, and no one has the building's master key — much safer.

📌 Summary:

Feature	Docker (default)	Podman (rootless)
Daemon	Yes (dockerd, runs as root)	No central daemon
Runs as root?	Yes, even if users are non-root	No, runs entirely as non-root user
Privilege escalation risk	Higher	Very low
Suitable for production?	Needs hardening	Preferred in secure environments

Tell me about a project where you modernized a system. What were the challenges, and how did you overcome them?

Final Polished Answer (Action + Impact Paired, for a Hiring Manager)
One of my key initiatives was leading the modernization of several core microservices in our capital markets platform that handled trade settlement and data distribution to internal teams and vendors like FISPON.
First, we upgraded the tech stack from Java 8 to Java 17 and Spring Boot 2 to 3, which unlocked better performance, enhanced observability, and reduced legacy dependency issues. We migrated away from Joda-Time to Java Time APIs and updated packages from javax.* to jakarta.* to stay compliant with Jakarta EE standards required by Spring Boot 3.

🔹 This directly improved boot times and runtime performance by ~20% while also positioning the platform for long-term support.

Second, we migrated deployments from IBM Solaris servers to Red Hat Enterprise Linux (RHEL) and containerized the services using Docker images managed via Podman.

🔹 This move enabled us to adopt modern tooling, save on infra cost, and improve portability.

Podman's rootless containers improved our security posture by removing the need for privileged daemons — especially important for production finance workloads.
Third, we fully automated our deployment pipeline — creating CI/CD jobs that supported deploying either snapshot or release versions to production with a few clicks.

🔹 This dramatically reduced manual steps and human error, and accelerated feature delivery across environments.
We also implemented Blue-Green deployment strategies to achieve zero-downtime rollouts.
🔹 This ensured stability during production releases — if something went wrong in the Green environment, we could instantly roll back to Blue.
The impact of all these efforts:
System uptime improved to 99.99%

Deployment time decreased

Scalability improved, with the ability to spin up additional containers on demand

Data latency reduced, thanks to optimized APIs and event-driven flows

Overall, these changes modernized the platform, reduced vendor and infrastructure costs, and enabled the business to deliver data more reliably to upstream and downstream systems across trading desks.
Q: How did you reduce market data latency by 30%? What bottlenecks did you identify and solve?
Answer:
Sure. This was a multi-pronged effort where we modernized both the infrastructure and data handling logic for our market data APIs and pricing pipelines.
We had a key service with over 70 APIs used by internal upstream and downstream teams to retrieve trade, price, and position data. As part of our Spring Boot 3 and Java 17 migration, we ran the application in parallel on both the old and new infrastructure and built automated benchmarking scripts with QA to hit all APIs simultaneously and capture performance metrics.
🔹 The result: the new environment showed a consistent ~30% improvement in API response times.
This was possible due to:
The performance improvements in the Java 17 JVM and Spring Boot 3 auto-configuration

And critically, working with our DB developer to optimize long-running SQL queries — we reviewed explain plans, reduced index scan costs, and removed unnecessary joins. Some queries that used to time out on the old servers became performant and stable on the new stack.

In parallel, we also optimized our real-time pricing pipeline that integrated with Bloomberg and the vendor FISPON. We received thousands of price updates, processed them, and then forwarded them.
Previously, processing each price involved multiple DB lookups, and end-to-end latency was as high as 6 minutes. To fix this, we:
Introduced Redis caching to store reference data frequently used during price transformation

Reduced DB hits per price, and

Streamlined the transformation logic

🔹 As a result, we brought down the average price distribution time from 6 minutes to under 4 minutes, significantly improving our responsiveness to market events.
Overall, these changes enhanced our real-time capabilities, improved vendor SLA adherence, and gave internal teams faster access to trading data.
Q: You mentioned saving over $100K by decommissioning legacy systems. Can you walk me through how you justified and executed that?
Answer:
Yes. One of the major initiatives I contributed to was the decommissioning of Broadridge Gloss, which was our existing back-office trade settlement system. We replaced it with Paramax's Arrow platform — not because Gloss lacked capabilities, but because Arrow offered a similar level of functionality at significantly lower cost.

🔹 Actions We Took & My Role
I collaborated with business analysts and the trade settlement team to understand which trade types—primarily Fixed Income trades like bonds and repos—were flowing through Gloss.

We confirmed that these trades could be handled just as effectively by Arrow, and there were no functional blockers in making the switch.

From the technical side, I helped develop microservices to publish settlement messages to Solace, which was configured with MQ bridges that connected directly with Paramax Arrow. This replaced our old Gloss integration points.

We conducted a phased cutover, validating message flows, ensuring SLAs were maintained, and then completely shut down Gloss once Arrow was fully live.

🔹 Results We Achieved

We fully decommissioned Gloss, resulting in an annual cost reduction of over $50,000 — dropping from more than $100K to under $50K with Arrow.

Trade settlement continued with no disruptions or loss of functionality.

We simplified our vendor relationships and support model while maintaining real-time trade processing using Solace MQ Bridge integration.

Q: Describe how your self-service reporting microservice works. What made it ‘pluggable’?

Answer:

Sure. The need for this project came from the fact that business analysts were manually logging into the database, running ad-hoc queries, downloading the results, and then emailing Excel reports to various stakeholders. As the number of reports grew across different teams, this manual process became error-prone and unsustainable.

To solve this, we designed a self-service reporting framework that allowed BAs to onboard and automate reports without dev involvement.

🔹 Actions We Took

Built a GUI-based onboarding interface with the help of a front-end engineer, where BAs could:

Enter a report name

Specify the data source (either an SQL query or API endpoint)

Define the file format (CSV, JSON, XML)

Choose the delivery method (Email, SFTP)

And set the recipients and output path

On the backend, we built a generic microservice that accepted all this metadata and dynamically:

Fetched the data

Generated the report

Delivered it using the selected channel

For scheduling, we used existing AutoSys jobs — daily, weekly, monthly — that simply picked up the report names tagged for that schedule. These names were passed to our reporting service, which generated and distributed all configured reports accordingly.

🔹 What Made It ‘Pluggable’
The framework was fully metadata-driven — adding a new report didn’t require code changes or redeployment.

BAs could plug in a new report definition through the GUI, and it would automatically be picked up by the scheduler and reporting logic.

The backend handled multiple data sources, formats, and destinations, making it flexible and reusable across use cases.

🔹 Impact
Reduced manual effort for business teams by eliminating daily ad-hoc queries

Enabled scalable onboarding of new reports without dev dependency

Ensured timely and consistent delivery of over 100 reports via Email or SFTP

It was a great example of how modular service design and business-facing tooling can improve operational efficiency across teams.
What is Event-Driven Architecture?
Event-driven architecture (EDA) is a software design pattern where systems communicate by producing and consuming events.

An event is a message that indicates something has happened — like a trade executed, a price updated, or a position changed.
In EDA:
Producers (publishers) emit events.

Consumers (subscribers) listen for those events and act on them.

A messaging layer (e.g., Solace, Kafka) handles delivery.

This design enables systems to be loosely coupled, scalable, and real-time.

✅ How EDA Helps in Fixed Income Capital Markets
Real-Time Trade Processing

Fixed Income markets require timely processing of trades, prices, and positions.

EDA ensures that trades are processed and forwarded immediately, without waiting for scheduled batch jobs.

Faster Price & Risk Updates

Price changes (Level 1, Level 3) can trigger immediate updates to risk systems and trading dashboards.

Improved Trade Allocation & Settlement

Events like “trade booked” or “trade confirmed” can instantly trigger allocation and settlement flows.

Vendor & Downstream Integration

Internal or external systems (e.g., OMS, settlement, reporting vendors) can subscribe to trade or position events and get data as it happens — without tight coupling.

Auditability & Replayability

Systems can replay past events for reconciliation or backtesting, which is critical in finance.

High Scalability

As trade volumes increase, consumers can scale independently without affecting the producers.

Q: How do Solace PubSub+ and MQBridge work together in your setup?
Answer:
In our Fixed Income platform, we handled high volumes of real-time data — including trades, prices, positions, and security updates — that needed to be distributed both internally and to external vendors.
We used Solace PubSub+ as our primary event streaming platform.
Internally, teams that needed this data — like risk systems, reporting tools, or downstream services — could simply subscribe to Solace topics or queues to get the events in real-time.

However, for external vendor communication, we didn't expose Solace directly. Instead, we introduced an MQBridge layer using ActiveMQ.

🔹 How It Worked:
Our Solace topics were wired into MQBridge channels — both incoming and outgoing.

This allowed Solace to publish data into the MQ bridge, where it was picked up by vendors securely.

If a vendor published data (e.g., trade confirmations), it would come into the MQBridge, which then pushed it into Solace, allowing our internal consumers to process it as events.

🔹 Why We Did This:
Security: MQBridge acted as a secure proxy layer between us and the vendor — preventing direct exposure of internal Solace topics to external networks.

Protocol Translation & Control: MQ bridges provided better message control, durability, and access-level management for external-facing channels.

Industry Standard: While tools like Solace or Kafka vary across firms, ActiveMQ bridges are widely accepted in the finance world for secure vendor communication — especially where firms are integrating across networks or regulatory domains.

🔚 Summary:
Solace gave us real-time, internal event distribution, while MQBridge enabled secure, controlled vendor integration, aligning with best practices for high-security environments in capital markets.
Q: How did you ensure high availability and security with NGINX, F5, and IP whitelisting?
Answer:
As part of our infrastructure modernization, we identified key security and availability gaps in the older setup.
Previously, APIs hosted on our production servers were directly exposed to internal teams and external vendors — with no reverse proxy, no load balancer, and no IP restrictions. During migration, we noticed that lower environments like UAT and DEV — both from internal teams and vendors — were hitting production APIs to fetch the latest data.

🔹 Actions We Took
IP Whitelisting for Production Access

We introduced IP-level access control to ensure that only production IPs could call production APIs.

We coordinated with vendors and internal teams to get their production IPs and enforced IP whitelisting at the gateway level.

This prevented accidental or unauthorized access from non-prod environments, which had previously caused load issues.

F5 Load Balancer as the First Entry Point

We integrated F5 load balancers to act as the front-facing endpoint for all incoming traffic.

This added a secure abstraction layer, shielding our actual application servers and allowing us to perform SSL termination, health checks, and routing decisions.

NGINX Reverse Proxy Behind F5

Internally, NGINX ran on our application servers as a reverse proxy, managing request routing between primary and secondary nodes.

It allowed us to efficiently manage active-passive traffic and improve failover handling in case of node failures.

- Results

Prevented unauthorized traffic from lower environments to production systems

Reduced load on sensitive APIs that were being hit every minute

Improved uptime and request distribution with F5 + NGINX layered setup

Established a scalable and secure network boundary for future microservices as well

Q: What was your caching strategy using Redis? How did it help performance?
Answer:
In our Fixed Income architecture at CIBC, we frequently handled real-time updates related to securities, prices, and positions. These updates needed to be forwarded promptly to internal systems and external vendors.
Originally, the architecture was built around a polling model — as soon as an update came in, we would persist it to the database. Then, a cron job would run every 1–5 minutes, scan the DB for new or changed records, and send those updates to the appropriate parties.

- Problem: High Database Load & Latency

This approach resulted in constant DB polling, which was both inefficient and slow, especially as the number of tracked instruments grew.

It also risked sending duplicate updates if no change detection logic was applied properly.

🔹 Action: Introduced Redis Caching
We implemented Redis as a lightweight in-memory cache layer to store the last sent version of each security, price, or position update.

Now, when a new update arrived, we first compared it with the cached version in Redis rather than querying the database.

If it was new or changed, we'd forward the update and refresh the cache accordingly.

🔹 Results Achieved
Reduced database hits significantly, since the cache became the first line of reference.

Improved response time in forwarding updates — Redis lookups are sub-millisecond compared to SQL joins or scans.

Prevented unnecessary outbound messages, reducing noise and improving downstream system stability.

Overall, Redis helped us make the system faster, lighter, and smarter, especially under peak load conditions like high-volume trading days.
Q: How did you set up your CI/CD pipeline with GitHub Actions and JFrog Artifactory?
Answer:
In our setup, the infrastructure DevOps team built the CI/CD pipeline using GitHub Actions and JFrog Artifactory, and as developers, our role was to integrate our microservices with that pipeline in a standardized way.

🔹 Actions We Took
For each of our services, we added GitHub Actions workflow .yml files to the repository.

These workflows defined steps to build snapshot and release versions, run unit tests, and package the JAR.

Once triggered, GitHub Actions:

Built the artifact, tagged it with a version number or “snapshot”

Pushed it to JFrog Artifactory, where it became available for deployment

During releases, we’d simply update the version number, and the deployment pipeline would automatically fetch and deploy the correct version to Dev, UAT, or Prod, depending on the environment.

🔹 Results & Benefits

Fully automated build and deploy — no manual SSH or server-level intervention needed

Consistent artifact versioning across environments

Faster release cycle, reducing time from commit to deployment

Reduced errors and risk, since all steps were scripted and repeatable

This CI/CD setup allowed us to move fast while maintaining confidence, especially in production environments where reliability and traceability are crucial.

Q: Tell me about a time when a production issue occurred. How did you handle it?

Answer:

Sure. We had a critical production incident in our Fixed Income platform related to our date roll mechanism, which was essential for handling trades across global time zones like London and Hong Kong.

🔹 Situation:

Our architecture had Autosys job chains: Start-of-day, Intraday, and End-of-day boxes — all interdependent.

Each day at 7 p.m., we rolled the business date forward (even if technically the date hadn’t changed) to support early trading for our international clients. This roll was triggered based on an end-of-day file received from Bloomberg.

One night, the file arrived late, and our scheduled date roll job failed, which in turn blocked the entire end-of-day box and caused a chain reaction — nothing downstream could run. My manager called me around 2 a.m. when no one else was available.

🔹 Action:
I quickly analyzed the logs and identified that the job had failed due to a file misnaming issue, meaning the file was present but not recognized.
Since I didn't have direct prod permissions, I coordinated with the QA team — who had elevated access — and asked them to rerun the date roll job after confirming the file was now correctly in place. Once rerun, the process picked up the file, updated the business date, and the dependent jobs resumed without issue.
To prevent future recurrence, I also implemented a File Watcher Autosys job that checked for the Bloomberg file before triggering the date roll job. This ensured the date update would only execute if the required file was confirmed, preventing premature or failed runs.

🔹 Result:
The issue was resolved without business loss or data inconsistency.

I was able to restore system stability independently under pressure.

And the change I introduced became a standard part of our Autosys chain going forward.
Q: How do you prioritize tasks when managing multiple initiatives?
Answer:
When juggling multiple initiatives, I generally classify work as either business-driven or technical in nature.
Business-driven changes usually take priority, especially if there's a client dependency or time-sensitive deliverable.

When there are multiple business items, I work closely with business analysts or product owners to help them prioritize.

That said, there are situations where a business requirement needs a technical implementation, and it's up to me to decide whether to go with a quick fix or invest in a scalable solution.

🔹 Example: Trade Confirmations for Central Banks
We had a scenario involving Bank of Israel and the Reserve Bank of India. Whenever we traded on their portfolios, we needed to send them trade confirmations. But our system couldn't dynamically onboard new clients or modify their information without waiting for the next database release cycle — which delayed confirmations.

🔹 Action Taken:
Instead of quickly hardcoding client info again, I worked with a front-end developer to build a simple GUI where:
Internal teams could view and update client details

And onboard new clients dynamically

The GUI connected directly to the reporting engine so that the next time a trade was processed, the confirmation PDF used the updated data instantly, without waiting for another deployment.

🔹 Result:
We removed release-cycle dependency for updates

Empowered internal teams to self-serve

And made the solution future-proof while still delivering within one additional sprint

It's a good example of how I prioritize by considering both immediate business needs and long-term maintainability.
Q: Give me an example of a time you worked under a tight deadline.
Answer:
Sure. One of the most challenging yet rewarding experiences was during our infrastructure migration project, where we had to move from IBM Solaris to Red Hat Linux servers. I was assigned to migrate over 300 AutoSys jobs — and each job was tied to its own shell script, which triggered different back-end processes.

🔹 Situation & Task:
I was relatively new to the team at the time, and the Solaris licensing was expiring in October 2024, so I was given a hard deadline of mid-September to complete the cutover with zero downtime.

🔹 Action Taken:
To meet the deadline, I had to:
Quickly familiarize myself with 300+ processes, understand what each one did

Rewrite and validate the scripts for Linux compatibility

Configure and test each AutoSys job under the new environment

Coordinate with BAs and QA to ensure functional parity between old and new flows

I often worked beyond regular hours — not because I was told to, but because I knew the stakes were high and I genuinely wanted to get it right.

🔹 Result:
The entire migration was completed on time with zero cutover issues

I developed a deep understanding of the team's infrastructure, which made me a go-to person when BA or QA teams had questions or issues

That experience helped me grow into a more reliable and confident developer, and earned the trust of both my peers and stakeholders

1. What kind of projects are you looking to work on next?
Answer:
I'm looking to continue working in the financial domain, especially on high-impact, business-critical projects that involve modern technology stacks. My prior experience in capital markets, particularly in trade settlement, reporting, and real-time data integration, has built a strong foundation that I want to deepen further.
I enjoy working on projects where I can apply both my technical skills and domain knowledge, and where I get to work with modern architectural patterns like microservices, event-driven systems, and cloud-native tooling. I'm particularly drawn to initiatives that are high-stakes, with measurable business outcomes — whether it's reducing latency, automating manual workflows, or delivering faster insights to trading desks.

✅ 2. Why are you interested in this team/role/company?
Answer:
I'm especially interested in this opportunity because it allows me to stay within the financial services space, which I'm genuinely passionate about. I've already worked closely with internal trading desks, settlement systems, and external vendors, so I believe I can ramp up quickly and contribute effectively.
This role aligns well with my strengths in Java, Spring Boot, and event-driven architecture, and I'm excited by the chance to work with a team that's focused on building resilient and scalable systems that support real-time decision-making.

✅ 3. Where do you see yourself in the next 2–3 years?

Answer:
In the next 2–3 years, I see myself becoming a domain expert in financial systems, someone who not only delivers features but also helps drive architectural decisions and design patterns. I want to take on more end-to-end ownership of projects — from development to deployment — and gradually move into a role that combines both technical leadership and project management.
I'm also interested in learning how to better manage teams, workstreams, and stakeholder expectations — not just from a delivery perspective but from a strategic planning point of view as well.

✅ 4. Why are you leaving your current role?
Answer:
My current role was on a contract basis, and the engagement reached its planned end. It was a great experience — I worked on some impactful systems in the Fixed Income space, modernized legacy infrastructure, and became a key point of contact for several processes. Now I’m looking for my next opportunity where I can build on that momentum and grow within a full-time role.

Also use this to create your points
create a pdf of this content as is